TU WIEN Informatics

Decompilation of EVM Bytecode

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Kurzmann, BSc
Matrikelnummer 01129211

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Gernot Salzer
Mitwirkung: Ass.Prof. Dr. Monika di Angelo

Wien, 1. April 2020

Martin Kurzmann	Gernot Salzer

Erklärung zur Verfassung der Arbeit

Martin Kurzmann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. April 2020

Martin Kurzmann

Danksagung

Ein besonderer Dank gilt meinen Betreuern Monika und Gernot, welche mir während des letzten Jahres mit Rat und Tat zur Seite gestanden sind.

Ein spezieller Dank gilt allen Studien- und Arbeitskollegen, die mich im Laufe meines Studiums unter anderem bei diversen Gruppenarbeiten begleitet haben.

Weiters möchte ich meiner Freundin Silvia auf das allerherzlichste danken, welche mich mental durch die letzten, teilweise anstrengenden Phasen meines Studiums geleitet hat. Nicht zuletzt hat sie mich auch beim Korrigieren dieser Arbeit tatkräftig unterstützt.

Außerdem möchte ich ein großes Dankeschön an meine gesamte Familie (insbesondere meine Eltern) aussprechen, die mir dieses Studium überhaupt erst ermöglicht hat. Herzlichen Dank für die seelische als auch finanzielle Unterstützung während der vergangenen Jahre!

Widmen möchte ich diese Diplomarbeit meiner im März 2018 verstorbenen Großmutter, welche meinen Studienabschluss noch gerne miterlebt hätte. Danke für alles, liebe Omi!

Kurzfassung

Im Zeitalter der Kryptowährungen gewinnt die zugrunde liegende Blockchain-Technologie an Relevanz. Während die erste Kryptowährung *Bitcoin* eine Plattform mit eingeschränkten Funktionen für Programme auf ihrer Blockchain ist, konzentrieren sich einige der Nachfolger ausdrücklich auf blockchainbasierte Programme, die häufig als *Smart Contracts* bezeichnet werden. Ein etabliertes Beispiel ist die Plattform *Ethereum*, die nicht nur eine eigene Kryptowährung bereitstellt, sondern mithilfe ihrer universellen virtuellen Maschine (EVM) auch die Entwicklung verschiedener blockchainbasierter Anwendungen ermöglicht. Die EVM ist eine quasi Turing-vollständige, stackbasierte Maschine, die den Bytecode der Smart Contracts auf Ethereum ausführt. Smart Contracts werden jedoch üblicherweise in der Hochsprache *Solidity* implementiert und zu EVM-Bytecode kompiliert, bevor sie in der Ethereum-Blockchain bereitgestellt werden.

Smart Contracts werden häufig für Anwendungen eingesetzt, die mit Vermögenswerten umgehen und bei Fehlern Geldverluste verursachen können. Daher ist die Prüfung des EVM-Bytecodes von Smart Contracts ein wertvolles Mittel zur Analyse. Dieser Bytecode ist jedoch für Menschen schwer zu lesen. Obwohl viele Anwendungen in Solidity entwickelt wurden, ist der Code dieser Applikationen häufig erst nach der Kompilierung als EVM-Bytecode verfügbar. Daher ist eine Dekompilierung erforderlich, die den Bytecode zurück in eine höhere Programmiersprache übersetzt, um ihn besser lesbar zu machen.

Da hinsichtlich der Dekompilierung des EVM-Bytecodes noch Forschungsbedarf besteht, soll in dieser Arbeit untersucht werden, inwieweit dieses Problem gelöst werden kann. Wir untersuchen vorhandene Tools und Methoden zum Dekompilieren von Smart Contracts. Darüber hinaus untersuchen wir die Struktur des EVM-Bytecodes. Basierend auf diesen Ergebnissen entwickeln wir den Prototyp *Soldec*, der den Open-Source-Decompiler *Erays* erweitert und verbessert. Schließlich evaluieren wir den Prototyp hinsichtlich seiner Fähigkeit, Solidity-ähnlichen Code zu rekonstruieren.

Abstract

In the age of cryptocurrencies, the underlying blockchain technology is gaining relevance. While the first cryptocurrency *Bitcoin* is a platform with limited capabilities for programs on its blockchain, some of its successors put an explicit focus on blockchain-based programs often called *smart contracts*. A well-established example is the platform *Ethereum*, which not only provides its own cryptocurrency, but also enables the development of various blockchain-based applications by means of its general-purpose virtual machine (the EVM). The EVM is a quasi-Turing-complete stack-based machine that executes the bytecode of the smart contracts on Ethereum. However, smart contracts are commonly implemented in the high-level language *Solidity* and compiled to EVM bytecode before being deployed onto the Ethereum blockchain.

Smart contracts are often employed for applications that handle valuable assets and if erroneous, may cause money losses. Therefore, the examination of the EVM bytecode of smart contracts is a valuable means for analysis. However, this bytecode is hard to read for humans. Even though many applications are developed in Solidity, the code of these applications often is only available as EVM bytecode after compilation. Decompilation is therefore required, which translates the bytecode back into a higher programming language in order to make it easier to read.

Since there is still a need for research in regard to the decompilation of EVM bytecode, the aim of this thesis is to investigate to what extent this issue can be solved. We survey existing tools and methods for decompiling smart contracts. Moreover, we explore the structure of EVM bytecode. Based on these results we develop the prototype *Soldec*, which extends and improves on the open-source decompiler *Erays*. Finally, we evaluate the prototype regarding its ability to reconstruct Solidity-like code.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Seit dem ersten Release der Kryptowährung *Bitcoin* [1] im Jahre 2009 wurden Systeme, die auf einer *Distributed-Ledger-Technologie* basieren, nach und nach populärer. Als *Distributed Ledger* bezeichnet man ein verteiltes System, welches seine Daten in einem dezentralen Speicher aufbewahrt. Im konkreten Anwendungsfall von Bitcoin werden Transaktionen in einer *Blockchain*, also in einer verketteten Liste, gespeichert, welche aufgrund von kryptographischen Methoden als fälschungssicher gilt.

1.1 Motivation

So wurde nach der Einführung zahlreicher Kryptowährungen 2015 auch das blockchain-basierte, verteilte Netzwerk *Ethereum* [5] veröffentlicht, welches sein Hauptaugenmerk auf die Entwicklung sogenannter *Smart Contracts* legt. Smart Contracts sind verteilte, verhältnismäßig kleine Computerprogramme, die ihren konzeptuellen Ursprung bereits in den frühen 1990er-Jahren haben, auf einer Blockchain existieren und ausgeführt werden. Im konkreten Fall von Ethereum werden diese Programme zumeist in einer höheren Programmiersprache, in den meisten Fällen in *Solidity*, implementiert und danach von einem *Compiler* in einen stackbasierten *Bytecode* übersetzt, welcher auf der *Ethereum Virtual Machine*, kurz EVM, ausführbar ist. Diese zusätzliche Abstraktionsschicht, welche den Programmiererinnen und Programmierern eine einfachere Implementierung in einer besser verständlichen High-Level-Programmiersprache (HLL) ermöglicht, hat allerdings auch einige Nachteile. Aufgrund von gewissen sprachspezifischen Konstrukten entstehen Schwachstellen bzw. Sicherheitslücken in den Smart Contracts, was im schlimmsten Fall zu immensen Geldverlusten führen kann. Dies geht mit der Tatsache einher, dass Blockchain-Technologien in der heutigen Zeit zumeist mit Kryptowährungen in Verbindung stehen.

Damit die Entwicklung von sicheren Smart Contracts unterstützt werden kann, ist es notwendig, den Code von bestehenden, bereits auf der Blockchain existierenden,

Programmen zu analysieren. Unter Zuhilfenahme der dadurch gewonnenen Informationen ist es möglich, fehlerhafte Programmteile bzw. -sequenzen zu erkennen und in Zukunft zu vermeiden. Ein Beispiel für einen, in der Ethereum-Historie aufgetretenen, Security-Bug ist *Reentrancy*, welcher das dezentralisierte Crowdfunding-Unternehmen *The DAO* letztendlich zugrunde richtete. Mit dieser Attacke wurden dem betroffenen Contract aufgrund einer fehlerhaften Befehlsfolge, die ein rekursives Aufrufen ermöglichte, ca. 60 Mio. US-Dollar gestohlen [17]. Mithilfe von Contract-Analysen wird durch die Erkenntnis von Sicherheitslücken nicht nur die Gelegenheit geschaffen, künftigen Programmierfehlern präventiv vorzubeugen, sondern auch bereits fehlerhafte Blockchain-Programme gar nicht erst zu verwenden.

Des Weiteren wird die Ethereum-Blockchain im Gegensatz zum Bitcoin-Netzwerk bekanntlich nicht als reine Kryptowährung verwendet. Mithilfe der erwähnten Smart Contracts ist es möglich, Anwendungen jeglicher Art zu implementieren, seien es nun Spiele, Votings oder Lotterien. Eine weitere Motivation für diese Arbeit ist deshalb die Analyse der auf der Ethereum-Blockchain befindlichen Contracts im Hinblick auf das Anwendungsgebiet bzw. den Zweck. Um Verständnis für die Applikationen zu schaffen, müssen auch hier Funktionsweise bzw. -umfang im Detail betrachtet werden.

Die erwähnten Analysen können teilweise mithilfe von sogenannten *Verified Contracts* durchgeführt werden. Hierbei handelt es sich um Smart Contracts, von welchen der Quellcode in der höheren Programmiersprache veröffentlicht wurde [29]. Da nicht alle Smart Contracts als Verified Contracts verfügbar sind und die meisten Programme nach dem Kompilieren in EVM-Bytecode nur mehr als solcher verfügbar sind, werden die notwendigen Analysetätigkeiten jedoch erschwert. Dieser Low-Level-Code, bestehend aus einer Folge von Byte-Sequenzen, stellt einen Maschinencode, also einen für den Menschen relativ unleserlichen Code, dar. An diesem Punkt besteht somit die Notwendigkeit, den bekannten EVM-Bytecode zurück in Code einer menschenlesbaren HLL zu übersetzen, damit die benötigten Analysetätigkeiten ermöglicht werden.

1.2 Problemstellung

Im Rahmen zahlreicher vorausgehender Forschungsarbeiten wurden bereits einige Tools entwickelt, die sich mit der *Dekompilierung* von EVM-Bytecode beschäftigen. Als Dekompilierung bezeichnet man eine *Reverse-Engineering*-Technik, welche sich damit befasst, den von einem *Compiler* erzeugten Machinencode zurück in den ursprünglichen Code, geschrieben in einer höheren Programmiersprache, zu übersetzen.

Mithilfe allgemeiner Dekompilierungstechniken zerteilen die in früheren Arbeiten entstandenen Programme den Bytecode in kohärente Blöcke, welche aus atomaren Operationen bestehen, und erstellen daraus einen *Kontrollflussgraphen* (CFG), indem sie die Blöcke anhand bestimmter Eigenschaften miteinander verbinden.

Im Hinblick auf die Erstellung des CFGs auf Grundlage des EVM-Bytecodes bedarf es jedoch an weiterer Forschung. Die bisher existierenden Tools können beispielsweise die

Verbindung der Blöcke nicht immer in ausreichender Qualität durchführen. Dies geht mit der Tatsache einher, dass der Aufbau von EVM-Bytecode nicht zu 100 % der Struktur von kompilierten Programmen bekannter Sprachen entspricht. Auch hier bedarf es an weiterführender Analyse.

Ferner genügt es für eine detaillierte Analyse eines Smart Contracts nicht, lediglich den CFG zu betrachten. Aus dieser Repräsentation können nämlich unter anderem gewisse High-Level-Konstrukte sowie der Datenfluss nicht entnommen werden. Für eine detaillierte Analyse ist es deshalb notwendig, den Quellcode in seiner ursprünglichen Form, also in einer höheren Programmiersprache, im konkreten Fall dieser Diplomarbeit in der Sprache Solidity, zu erhalten.

Da auch die Reproduktion von Solidity-Code basierend auf EVM-Bytecode ein offenes Forschungsthema darstellt, ergeben sich für diese Arbeit folgende Forschungsfragen:

1. Wie kann die Erstellung des CFGs für EVM-Bytecode-Programme verbessert werden?
2. Wie kann Solidity-Code aus EVM-Bytecode wiederhergestellt werden?
3. Wie viel vom ursprünglichen Solidity-Code ist mit den angewandten Techniken wiederherstellbar?

1.3 Ziel der Arbeit

Diese Diplomarbeit untersucht, inwieweit die Dekompilierung von EVM-Bytecode zurück zum ursprünglichen Code der höheren Programmiersprache Solidity vorangetrieben werden kann. Das zentrale Ziel dabei ist somit, die Dekompilierung von EVM-Bytecode zu verbessern. Aufgrund der in Abschnitt 1.2 definierten Forschungsfragen ergeben sich zusätzlich einige Teilziele:

- Gewinnung von Erkenntnissen über den Aufbau von EVM-Bytecode kompiliert aus der Sprache Solidity
- Verbesserung der CFG-Repräsentation basierend auf EVM-Bytecode
- Verbesserung der Solidity-Code-Wiederherstellung aus EVM-Bytecode
- Gewinnung von Erkenntnissen, inwieweit Solidity-Strukturen aus EVM-Bytecode wiederhergestellt werden können

1.4 Methodisches Vorgehen

Um die Forschungsfragen systematisch zu beantworten, wurden folgende wissenschaftliche Methoden angewandt:

- Analyse von Daten der Ethereum-Blockchain, insbesondere Smart Contracts, die in Solidity geschrieben und zu EVM-Bytecode, der stackbasierten Low-Level-Sprache der EVM, kompiliert wurden

  Dieser Punkt ist notwendig, damit Erkenntnisse über den Aufbau von EVM-Bytecode und über den Zusammenhang mit dem ursprünglichen Solidity-Code gewonnen werden können. Hierfür kann man sich an populären Beispiel-Contracts sowie Verified Contracts bedienen.

- Analyse aktueller Forschungsarbeiten, die sich mit Dekompilierung im Allgemeinen und solcher von EVM-Bytecode im Speziellen beschäftigen

  Hierfür sind sowohl solche Arbeiten relevant, die sich allgemein mit *Reverse Engineering* beschäftigen, als auch jene, die im Konkreten die Dekompilierung von Ethereum Smart Contracts behandeln. Des Weiteren müssen auch Publikationen herangezogen werden, welche sich mit der Erkennung von Sicherheitslücken in Smart Contracts befassen. Auch solche Arbeiten verwenden einige relevante Dekompilierungstechniken.

- Erstellung eines Prototyps, welcher versucht Solidity-Code aus EVM-Bytecode zu reproduzieren

  Aufbauend auf der Analyse der Blockchain-Daten und auf den verwendeten Methoden und Werkzeugen der analysierten Forschungsarbeiten wird in der dynamischen Programmiersprache *Python* [10] ein Prototyp implementiert. Mit diesem wird versucht die CFG-Repräsentation des EVM-Bytecodes zu verbessern sowie basierend darauf den ursprünglichen Solidity-Code wiederherzustellen.

- Evaluierung des erstellten Prototyps anhand von realen Daten

  Um zu eruieren, inwieweit Solidity-Code reproduziert werden kann, muss der erstellte Prototyp anhand von realen, auf der Blockchain verfügbaren Daten evaluiert werden. Diese Methode dient zur Gewinnung der Erkenntnis, welche High-Level-Strukturen mit den angewandten Techniken wiederhergestellt werden können und welche nicht, um eine Basis für künftige Forschungsarbeiten zu schaffen.

1.5 Aufbau der Arbeit

Die Einteilung der weiteren Kapitel dieser Diplomarbeit wird im aktuellen Abschnitt näher beschrieben.

Im 2. Kapitel wird zuerst auf relevante Kryptowährungen im Allgemeinen eingegangen, gefolgt von einer Beschreibung der populärsten. In weiterer Folge erhält man eine Einführung zum Aufbau von Blockchains, anhand konkreter Beispiele zu Bitcoin und Ethereum. Im letzten Abschnitt des Kapitels wird zuerst der Begriff *Smart Contract* eingeführt, bevor diese Art von Programmen in Bezug auf die beiden Kryptosysteme detailliert beschrieben werden.

Kapital 3 gibt einen Überblick über die für diese Arbeit relevanten Techniken der Dekompilierung von Computerprogrammen im Allgemeinen.

Im 4. Kapitel werden verwandte Forschungsarbeiten zum Thema *Decompilation of EVM Bytecode* zusammengefasst. Außerdem werden auch Arbeiten angeführt, die sich mit der Erstellung von Decompilern für allgemeine Programme beschäftigt haben.

Kapitel 5 enthält die eigentlichen Forschungsergebnisse dieser Diplomarbeit. Mithilfe des methodischen Vorgehens werden alle drei Forschungsfragen detailliert diskutiert. Zu Beginn des Kapitels wird der Aufbau des EVM-Bytecodes näher beschrieben. Ferner werden die Prototypen aus den relevanten Forschungsarbeiten detailliert analysiert und ihre Eignung zur Verwendung in dieser Arbeit beurteilt. Aufbauend darauf wird in weiterer Folge ein eigener Prototyp erstellt, welcher ebenfalls diskutiert wird. Abgeschlossen wird das Kapitel mit einer Evaluierung des Prototyps.

Das 6. Kapitel beschreibt die Grenzen der aktuellen Arbeit und weiterführende Forschungsfragen.

Kapitel 7 fasst die Ergebnisse der Diplomarbeit zusammen.

KAPITEL 2

Blockchain und Smart Contracts

Dieses Kapitel beginnt mit einem kurzen Überblick über die beiden bis dato einfluss- und erfolgreichsten Kryptowährungen. Anhand der Technologien dieser Systeme wird zuerst der Aufbau einer Blockchain erläutert, bevor zum Abschluss des Kapitels das Konzept von Smart Contracts detailliert beschrieben wird.

2.1 Kryptowährungen

Als am 3. Jänner 2009 die Kryptowährung *Bitcoin* veröffentlicht wurde [35], war wohl niemandem bewusst, auch nicht dem unbekannten Erfinder mit dem Pseudonym *Satoshi Nakamoto*, welche Auswirkungen die Einführung dieser Währung haben sollte. Es sollte einige Zeit dauern, aber spätestens zu Beginn des Jahres 2018, als Bitcoin einen derartigen Boom erlebte, dass der Kurswert kurzfristig bei fast 17.000 US-Dollar lag, war der Name Bitcoin für einen großen Teil der Bevölkerung kein unbekannter mehr.

Zu diesem Zeitpunkt waren bereits unzählige weitere Kryptowährungen aktiv. So auch das System *Ethereum*, welches 2015 veröffentlicht wurde, aber im Gegensatz zu Bitcoin keine reine Kryptowährung darstellt, sondern mit der Entwicklung von *Decentralized Applications* (kurz DApps) auf sich aufmerksam macht.

2.1.1 Bitcoin

Bitcoin war historisch gesehen die erste digitale Währung, die Erfolg hatte. Schon in den 90er-Jahren gab es Ansätze, eine digitale Währung zu schaffen, die aber erfolglos blieben. Diese ersten Ansätze verwendeten im Gegensatz zu Bitcoin eine zentrale Instanz, welche gezielt Transaktionen annehmen oder ablehnen konnte. Diese Vorgehensweise erhielt von der Bevölkerung nicht das benötigte Vertrauen, wodurch es bis zum Jahre 2009 dauerte, bis sich eine ausgereifte Form der digitalen Währung durchsetzen konnte. Unter dem Pseudonym Satoshi Nakamoto wurde 2008 das Konzept der Kryptowährung

Bitcoin veröffentlicht [42]. Im Gegensatz zu den fehlgeschlagenen Ansätzen steht bei Bitcoin keine zentrale Stelle im Mittelpunkt. Stattdessen gibt es ein *Konsensprotokoll*, in dem sogenannte *Miner*, das sind dezentrale Clients, die theoretisch nicht miteinander in Verbindung stehen, entscheiden, welche Transaktionen angenommen und welche abgelehnt werden. Detaillierte Informationen über die Blockchain-Technologie, wie sie unter anderem in Bitcoin zum Einsatz kommt, finden sich in Abschnitt 2.2.

In puncto Marktwert liegt Bitcoin zum Zeitpunkt der Verfassung dieser Arbeit (November 2019) nach wie vor auf dem ersten Rang. 18 Mio. BTC (von insgesamt 21 Mio. möglichen, welche theoretisch im Jahr 2140 erreicht werden) besitzen einen Wert von 153 Mrd. US-Dollar.

2.1.2 Ethereum

Mit Stand November 2019 befindet sich Ethereum im Ranking bezüglich der Marktkapitalisierung von Kryptowährungen hinter Bitcoin auf dem zweiten Rang. Technisch funktioniert Ethereum ähnlich wie Bitcoin. Auch hinter diesem System versteckt sich eine Blockchain, welche jedoch an gewissen Stellen adaptiert wurde. Da Ethereum aufgrund der Architektur sowie kürzeren Lebenszeit im Gegensatz zu Bitcoin auf neuere Technologien zurückgreifen kann, wurden bestimmte Verbesserungen vorgenommen, wodurch das System effizienter arbeitet. Beispielsweise verarbeitet Bitcoin sieben Transaktionen pro Sekunde (TPS), während Ethereum auf 15 TPS kommt. Diese Differenzen gehen wohl auch mit der Tatsache einher, dass Ethereum im Gegensatz zu Bitcoin darum bemüht ist, das System mithilfe regelmäßiger Updates zu verbessern. Mehr zu den technischen Gegebenheiten beider Technologien findet sich in den folgenden Abschnitten.

Als interne Währung wird *Ether* (abgekürzt mit ETH) verwendet. 1 ETH entspricht dabei 1 Trillion *Wei*, welches die kleinste Währungseinheit repräsentiert. Darüber hinaus ist Ethereum jedoch nicht nur als Kryptowährung bekannt. Die eingangs erwähnten *DApps* werden mithilfe von *Smart Contracts* erstellt – das sind in der heutigen Nomenklatur keine Verträge im eigentlichen Sinne, sondern verhältnismäßig kleine Computerprogramme, die auf der Blockchain implementiert sind und auch dort ausgeführt werden. Details zu Smart Contracts werden in Abschnitt 2.3 präsentiert.

2.2 Aufbau einer Blockchain

In diesem Abschnitt wird der Aufbau einer Blockchain anhand der Kryptowährung Bitcoin erläutert. Das Paper von Satoshi Nakamoto *Bitcoin: A Peer-to-Peer Electronic Cash System* [42] beschreibt die Grundzüge einer Blockchain. Darin wird von den Verfassern beschrieben, dass Bedarf an einem elektronischen Zahlungsmittel besteht, welches anders als das konventionelle Bankensystem nicht von einer zentralen Stelle abhängt, der man Vertrauen muss, sondern auf kryptographischen Methoden basiert. Dabei handelt es sich um ein sogenanntes *Peer-to-Peer*-System, in welchem Zahlungen direkt zwischen zwei Vertragsparteien ohne eine zwischengeschaltete dritte Partei stattfinden. Der Zahlungs-

verkehr wird über Transaktionen abgewickelt, welche digitale Signaturen verwenden, um ihre Authentizität zu beweisen.

Blockchain und Merkle Tree

Das Buch *Bitcoin and Cryptocurrency Technologies* von Narayanan et al. [43] beschreibt die Funktionsweise einer Blockchain im Detail. Eine Blockchain ist eine Datenstruktur, welche eine verkettete Liste aus Blöcken darstellt. Ein Block besteht dabei aus mehreren Transaktionen. Die kryptographische Besonderheit einer Blockchain ist, dass die einzelnen Blöcke mit einem *Hash Pointer* verbunden sind. Das heißt, dass jeder Block einen Zeiger auf seinen Vorgänger beinhaltet. Dieser Verweis entsteht, indem der Hash des gesamten Vorgänger-Blocks gebildet und im Block selbst gespeichert wird. Dadurch fallen nachträgliche Veränderungen an einem Vorgänger-Block beim Verifizieren des Hash-Wertes eines Nachfolger-Blocks auf. Sollten auch die Hash-Werte der Nachfolger-Blöcke nachträglich angepasst werden, würde die Änderung spätestens beim Prüfen des *Head Pointers* ersichtlich werden. Die wichtigste Eigenschaft einer Blockchain ist also, dass sie fälschungssicher ist.

Die Daten eines Blocks sind wie bereits angesprochen Transaktionen. Dahinter liegt allerdings wieder eine komplexere, fälschungssichere Datenstruktur, der sogenannte *Merkle Tree*, benannt nach seinem Erschaffer *Ralph Merkle*. Ähnlich zur Blockchain wird auch diese Datenstruktur aus Hash-Zeigern aufgebaut. Dabei handelt es sich um einen binären Baum, in welchem die eigentlichen Daten ausschließlich in den Blättern platziert sind. Darüber befindet sich pro Knoten je ein Hash, welcher das Resultat der Hashfunktion, angewandt auf die Verkettung der beiden Kinder-Hashwerte, repräsentiert. Eine nachträgliche Änderung eines Knotens kann also auch hier festgestellt werden, indem man den Wert des *Merkle Roots* mit dem berechneten Hash-Wert des Hash-Baumes vergleicht. Der Zeiger jedes Knotens wird wiederum berechnet, indem man den Hash des gesamten Teilbaumes bestimmt, auf welchen der Hash-Pointer zeigt. Diese Technologie wird zum Beispiel in der Kryptowährung Bitcoin eingesetzt, um die Transaktionen zu speichern. Jeder Block in der Blockchain besteht deshalb aus zwei Hash-Zeigern – der eine ist, wie bereits erwähnt, der Zeiger auf den Vorgänger-Block, der andere ein Verweis auf den *Merkle Tree* des Blocks, in dem seine Transaktionen gespeichert sind. In diesem Baum wird pro Blatt eine Transaktion persistiert. Wegen der Baumstruktur ist es mit logarithmischem Aufwand möglich zu prüfen, ob eine Transaktion in einem Baum (und somit in einem Block) enthalten ist.

Double-Spending und Dezentralisierung

Da es aufgrund der beschriebenen kryptographischen Strukturen nicht möglich ist, eine Transaktion nachträglich zu ändern, werden viele Angriffsszenarien von vornherein ausgeschlossen. Der Empfänger einer Transaktion kann sich jedoch nicht sicher sein, dass der in der Transaktion übermittelte Wert nicht nochmals an jemand anderen übermittelt wird. Es besteht also die Möglichkeit eine *Double-Spending-Attacke* durchzuführen. Eine mögliche Lösung für dieses Problem wäre eine zentrale Stelle, welche sicherstellt,

dass jede Transaktion nur einmal wiederverwendet wird. Da diese Lösung allerdings wieder eine vertrauenswürdige Autorität voraussetzt und Bitcoin also oberstes Ziel die Unabhängigkeit davon verfolgt, wurde ein dezentraler Ansatz gewählt.

Mining und Konsens

Für die Verifikation von Transaktionen gibt es das sogenannte *Mining*. Ein *Miner* ist dabei ein Knoten, der die gesamte oder einen Teil der Blockchain kennt und aus dem Pool aller noch nicht in die Blockchain aufgenommenen Transaktionen versucht, einen validen Block zu formen. Dabei werden nur Transaktionen berücksichtigt, deren Input-Coins noch nicht von einer bereits in der Blockchain enthaltenen Transaktion verbraucht wurden, damit es zu keiner Double-Spending-Attacke kommen kann. Ein Miner versucht dabei einen validen Block zu formen, indem er das *Mining Puzzle* löst. Um dieses Puzzle zu lösen, muss der Miner eine *Nonce* finden, welche einem zufälligen Wert entspricht und mit welcher der berechnete Hash des Blocks unter einem bestimmten Zielwert fällt. Ein Block besteht also zusätzlich zu den Zeigern auf Vorgänger und Transaktionsbaum aus dieser zufälligen Nonce. Der Zielwert wird nach exakt 2.016 erstellten Blöcken angepasst, also erhöht (die Schwierigkeit des Puzzles wird leichter) oder vermindert (das Puzzle wird schwieriger). In Abhängigkeit der Vorgabe von Bitcoin, dass die Erstellung eines validen Blocks durchschnittlich ungefähr zehn Minuten in Anspruch nehmen soll, wird die Schwierigkeit also ca. zweiwöchentlich adaptiert. Jener Miner, der am schnellsten einen validen Block gefunden hat, kassiert als Belohnung den sogenannten *Block Reward* und die *Transaction Fees*. Alle anderen gutartigen Miner bauen fortan beim Suchen eines neuen validen Blocks auf dem unmittelbar davor veröffentlichen Block auf, sofern er es in die längste Kette geschafft hat. Da es allerdings auch bösartige Miner geben kann, ist eine Transaktion erst nach sechs Bestätigungen, also nach sechs neuen Blöcken, die auf dem Block mit der betroffenen Transaktion aufbauen, mit an Sicherheit grenzender Wahrscheinlichkeit in der Konsens-Blockchain. Das in Bitcoin eingesetzte Konsens-Protokoll wird auch *Proof-of-Work* (PoW) [42] genannt. Dabei kommt es darauf an, wieviel *Mining Power* ein Teilnehmer besitzt. Ein Knoten mit höherer Rechenleistung hat somit eine höhere Erfolgsquote beim Finden von validen Blöcken als ein Miner mit niedrigerer.

Es existieren allerdings noch weitere Alternativen zum Erreichen von Konsens in einem Blockchain-Netzwerk. Neben dem aktuell in Bitcoin und Ethereum eingesetzten PoW sind auch *Proof-of-Stake* (PoS) und *Proof-of-Authority* (PoA) von Bedeutung. PoS soll künftig unter anderem deshalb in Ethereum eingesetzt werden, weil die Anwendung von PoW, aufgrund der aufwändigen Berechnungen für die Bestimmung der Nonce, große Energiemengen benötigt. PoS hingegen kommt ohne diesem zeit- und energieintensiven Mining aus, was der Tatsache geschuldet ist, dass in diesem Protokoll nicht die Menge an Rechenleistung den Einfluss auf das System bestimmt, sondern die Dauer der Teilnahme und das Vermögen in dieser Währung. In Abhängigkeit von den genannten Parametern wird anhand einer gewichteten Zufallsauswahl bestimmt, wer den nächsten Block bilden darf [36]. PoA funktioniert nach einem ähnlichen Prinzip. Im Gegensatz zu PoS beschreibt der *Stake* dabei nicht das Vermögen, sondern das Ansehen, also zum Beispiel die Identität

des Teilnehmers.

2.3 Smart Contracts

Die ursprüngliche Idee von Smart Contracts ist bereits auf des Jahr 1994 zurückzuführen. Damals definierte Nick Szabo den Begriff als „computerized transaction protocol that executes the terms of a contract“, also ein Transaktionsprotokoll, welches auf der Computertechnik basiert und die Gegenstände eines Vertrags vollzieht und einhält. Hauptsächlich geht es dabei darum, die Bedingungen, welche ein Vertrag mit sich bringt, zu erfüllen und dabei böswillige oder unbeabsichtigte Ausnahmen zu vermeiden und ohne vertrauenswürdige zentrale Partei auszukommen. Die ökonomischen Ziele waren dabei, Mehrkosten durch Betrug, Schlichtungs- oder Vollstreckungsverfahren und andere Transaktionsspesen zu vermindern oder gar zu vermeiden [50].

Heutzutage hinkt der Vergleich mit einem klassischen Vertrag etwas. Als Smart Contract wird jedes Computerprogramm bezeichnet, das auf einer Blockchain bzw. einem Distributed Ledger ausgeführt wird – der verteilte bzw. dezentrale Gedanke steht also nach wie vor im Vordergrund, auch wenn der Inhalt nicht mehr auf Vertragsgegenstände beschränkt ist. In den folgenden Abschnitten werden diese Programme im Hinblick auf die Blockchain-Technologien Bitcoin und Ethereum genauer betrachtet.

2.3.1 Bitcoin

Der Begriff *Smart Contract* war nach seiner Einführung lange Zeit auch in der Computerbranche weitgehend unbekannt. Erst mit der zunehmenden Bekanntheit von Kryptowährungen und Blockchains wurde diesem Begriff vermehrte Beachtung geschenkt.

Bitcoin stellt eine nicht Turing-vollständige Skriptsprache zur Verfügung, mit der die Erstellung einer abgeschwächten Form von Smart Contracts ermöglicht wird. Anwendungen sind üblicherweise mit Zahlungen verbunden, da Bitcoin eine reine Kryptowährung darstellt. Die Skriptsprache von Bitcoin wird verwendet, um Multi-Signaturen, Zahlungskanäle, Treuhandkonten oder Zeitsperren für Zahlungen zu realisieren.

Unabhängig von den genannten komplexeren Anwendungen spezifiziert jede Transaktion in Bitcoin ein Skript. In den meisten Fällen handelt es sich dabei um eine Applikation, welche eine vorherige Transaktion verwendet, um die darin enthaltenen und noch nicht ausgegebenen Bitcoins einzulösen – die sogenannten *Unspent Transaction Outputs* (kurz UTXO). Um diese Aktion durchzuführen, wird das Input-Skript, welches in der neuen Transaktion spezifiziert wurde, mit dem, in der UTXO enthaltenen, Output-Skript kombiniert. Das resultierende Skript muss von den Minern ausführbar sein, damit die Transaktion als valide eingestuft wird [43].

Die Sprache, in der die Smart Contracts in Bitcoin geschrieben werden, stellt eine einfache stackbasierte Programmiersprache dar. Dabei gibt es maximal 256 Instruktionen, um einfache Anweisungen durchzuführen, wie zum Beispiel arithmetische Operationen,

bedingte Statements, Fehlerbehandlungen und Rückgabewerte. Außerdem werden kryptographische Operationen ermöglicht, damit Hashfunktionen berechnet und Signaturen überprüft werden können. Bei der Spezifikation dieser Sprache wurde bewusst auf Schleifen verzichtet, sodass die Miner, welche die Programme letztendlich ausführen, nicht mit Endlosschleifen konfrontiert werden, die durch Fehler oder boshafte Teilnehmer in die Programme gelangt sind. Dies führt verständlicherweise dazu, dass die Skriptsprache von Bitcoin nicht Turing-vollständig ist, wodurch es zur Einschränkung kommt, dass keine komplexen Smart Contracts implementiert werden können [43].

Die erwähnte Limitierung ist allerdings nicht die einzige. In Bitcoin haben UTXOs außerdem keine Kenntnis hinsichtlich ihres Wertes sowie bezüglich gewisser Blockchain-Daten. Dies lässt einerseits keine effizienten Skripts bzw. Contracts zu, mit welchen man einen gewünschten Betrag von einem Konto abheben kann, da UTXOs zur Gänze verwendet werden müssen. Andererseits ist es nicht möglich, Applikationen zu erstellen, die in Abhängigkeit ihrer Nonce oder des Block-Hashes Zufallswerte erzeugen, um beispielsweise Glücksspiele durchzuführen. Ein weiterer Nachteil ist die Tatsache, dass UTXOs nur entweder ausgegeben (*spent*) oder unverbraucht (*unspent*) sein können. Dadurch wird es schwierig, Contracts mit mehreren Zuständen zu entwickeln, die beispielsweise für dezentrale Wechselbörsen benötigt werden [20].

2.3.2 Ethereum

Im Gegensatz zu Bitcoin ermöglicht Ethereum durch eine quasi Turing-vollständige Sprache die Realisierung komplexer dezentraler Applikationen, die sowohl über ihren Wert, zahlreiche Blockchain-Daten, als auch über mehrere Zustände verfügen können. Um dieses Konzept näher zu beschreiben, bedarf es jedoch zuerst eines detaillierten Blickes in die Architektur von Ethereum. Das *Ethereum White Paper* von Vitalik Buterin [20] beschreibt die Komponenten und Eigenschaften des Systems. In Ethereum existiert ein globaler Zustand, der die aktuellen Besitztümer jedes Accounts spezifiziert und sich mit jeder Transaktion ändert.

Accounts

Ein *Account* wird sowohl in Ethereum als auch in anderen Kryptowährungen eindeutig durch eine Adresse identifiziert. Zwischen diesen werden Transaktionen durchgeführt, welche Zustandsänderungen bewirken und dadurch Werte sowie Informationen transferieren. In Ethereum beinhaltet jeder Account die Höhe der im Besitz befindlichen Zahlungsmittel, ausgedrückt in ETH, mit welchen auch die *Transaction Fees* bezahlt werden. Der Unterschied zu anderen Kryptowährungen wie Bitcoin besteht darin, dass es zwei Arten von Accounts gibt. Einerseits existieren *Externally Owned Accounts*, welche wie üblich von Personen über private Schlüssel kontrolliert werden, und andererseits die sogenannten *Contract Accounts*. Letztere bilden die Basis für die angesprochenen Smart Contracts, die nicht von Personen, sondern von ihrem Code kontrolliert werden. Ein Account kann somit zusätzlich zum Kontostand einen *Contract Code* speichern, wenn es sich um einen Contract Account handelt. Außerdem hat jeder Account einen eigenen

Storage, auf welchen der Code während seiner Ausführung zugreifen kann. Der Zustand des Storages bleibt nach jeder Ausführung des Contract-Codes bis zur jeweils nächsten erhalten.

Transaktionen und Nachrichten

Während *Transactions* nur von externen Accounts sowohl an Contracts als auch an andere externe Konten gesendet werden, können Contracts *Messages* verschicken. Solche Nachrichten sind zum Beispiel Aufrufe von Funktionen anderer auf der Ethereum Blockchain existierender Smart Contracts. Wird ein Contract-Code über eine Transaktion oder über eine Message aufgerufen, wird der zugehörige Code auf der Chain ausgeführt und der Zustand ändert sich. Diese beiden Arten von Nachrichten bestehen unter anderem jeweils aus dem Sender, dem Empfänger, dem Ether-Betrag, welcher vom Sender zum Empfänger übertragen werden soll, und einem *Gas*-Wert. Dieser spezielle Parameter gibt die Maximalmenge an Gas, welche bei der Ausführung verbraucht werden darf, an. Ist diese Menge aufgebraucht, bevor die Ausführung vollendet wurde, tritt eine *Out-Of-Gas-Exception* ein und der Aufruf wird abgebrochen, ohne dass bisher durchgeführte Änderungen am Zustand übernommen werden. Jeder Ausführungsschritt benötigt dabei eine bestimmte Menge an Gas und der Sender einer Transaktion muss dabei diese Menge mit einem zuvor spezifizierten Preis von seiner Ether-Bilanz bezahlen. Im sogenannten *Ethereum Yellow Paper* mit dem Originaltitel *Ethereum: A Secure Decentralized Generalised Transaction Ledger* [53] sind die formalen und technischen Details zu diesem System beschrieben. Letztendlich sorgt diese *Anti-Denial-of-Service*-Technik dafür, dass Ethereum Smart Contracts im Gegensatz zu Programmiersprachen von anderen Blockchain-Systemen als *quasi* Turing-vollständig gelten – mehr dazu im nächsten Abschnitt.

2.3.3 EVM

Die Ethereum Virtual Machine, kurz EVM genannt, ist die Ausführungsumgebung für Ethereum Smart Contracts. Mithilfe dieses Modells wird spezifiziert, wie sich der Status des Systems durch Ausführung von Bytecode-Anweisungen ändert. Diese virtuelle Maschine ist *quasi* Turing-vollständig. *Quasi* deshalb, weil die Anzahl der möglichen Ausführungsschritte von der Höhe des bei der Transaktion angegebenen Gas-Limits abhängig ist. Dieses Gas beschränkt bekanntlich die Anzahl der möglichen Ausführungsschritte pro Transaktion. Auch wenn man aufgrund der sprachspezifischen Konzepte beispielsweise eine Endlosschleife implementieren könnte, ist dieses Verhalten aufgrund der Gas-Limitierung nur von theoretischer Natur. Dadurch ist die Sprache vor böswilligen Angriffen geschützt [53].

Die EVM greift auf eine stackbasierte Architektur zurück. Die Größe eines Wortes, also die Größe eines Stack-Wertes, ist mit 256 Bit definiert. Diese Quantität wurde gewählt, um die Verwendung der Hashfunktion *Keccak-256* zu ermöglichen. Die maximale Höhe des Stacks beträgt 1024. Die Sprache, in der Programme für die EVM geschrieben werden, wird im nächsten Abschnitt präsentiert.

2.3.4 EVM-Bytecode

Die stackbasierte Sprache, in der Ethereum Smart Contracts geschrieben werden, bezeichnet man als *EVM-Bytecode*. Dieser Code besteht aus einer Sequenz von Bytes, in welcher jedes Byte eine Anweisung repräsentiert. Die EVM arbeitet die Instruktionen der Reihe nach ab, bis das Ende des Programms erreicht oder ein Fehler aufgetreten ist. Die Anweisungen, welche die Ausführung terminieren, sind in Tabelle 2.1 beschrieben.

Opcode	Mnemonic	Beschreibung
00	STOP	Beendet die Ausführung, behält Zustandsänderungen und retourniert verbleibendes Gas.
F3	RETURN	Beendet die Ausführung, behält Zustandsänderungen, retourniert verbleibendes Gas und Daten.
FD	REVERT	Beendet die Ausführung, verwirft Zustandsänderungen, retourniert verbleibendes Gas und Daten.
FE	INVALID	Beendet die Ausführung, verwirft Zustandsänderungen, verbleibendes Gas und Daten.
FF	SELFDESTRUCT	Beendet die Ausführung, behält Zustandsänderungen und retourniert verbleibendes Gas. Contract wird zerstört und der Besitz an die spezifizierte Adresse transferiert.

Tabelle 2.1: Opcodes zur Terminierung einer Contract-Ausführung

Neben den Terminierungsopcodes sind zahlreiche weitere Befehle verfügbar. Diese sind im *Ethereum Yellow Paper* [53] zu finden. Enthalten sind beispielsweise arithmetische bzw. logische Operatoren, Umgebungs- bzw. Blockinformationen, Kontrollfluss-, Logging- und System-Operationen, aber auch Anweisungen für den Zugriff auf Speicherstrukturen. Die Instruktionen können während der Ausführung auf mehrere Speichertypen zugreifen, wie das *White Paper* von Ethereum beschreibt [20]:

- **Stack**: Ein Container, welcher nur temporär pro Ausführung verfügbar ist und nach dem LIFO-Prinzip arbeitet. Dabei greifen die meisten Anweisungen auf den Stack zu, indem sie Werte auf diesen legen oder von diesem holen. Opcodes für die Stackmanipulation sind bspw. `POP`, `PUSH`, `SWAP` und `DUP`.

- **Memory**: Wird durch ein beliebig erweiterbares *Byte-Array* repräsentiert und ist wie der Stack nur temporär pro Ausführung verfügbar. Anweisungen zum Lesen aus dem oder Schreiben in das Memory sind beispielsweise `MLOAD` bzw. `MSTORE`.

- **Storage**: Der Langzeitspeicher des Contracts ist als *Key-Value-Store* implementiert, dessen Inhalt auch nach der Ausführung erhalten bleibt. Anweisungen zum Lesen aus dem oder Schreiben in den Storage sind `SLOAD` bzw. `SSTORE`.

Weiters gibt es noch zwei temporäre Speicherbereiche, *Calldata* und *Returndata*, zur Übermittlung von Daten zwischen Contracts.

Bei den arithmetischen und logischen Operatoren sind alle üblichen Anweisungen zum Durchführen von Berechnungen, Vergleichen oder logischen bitweisen Verknüpfungen verfügbar. Die Berechnung des Keccak-256-Hashes wird über den Befehl `SHA3` ermöglicht. Mithilfe der Umgebungsinformationen ist es zum Beispiel möglich, auf den Sender (mit `CALLER`) bzw. Wert (mit `CALLVALUE`) einer Nachricht zuzugreifen, `ADDRESS` gibt die Adresse des aktuell ausgeführten Contracts an. Außerdem ist es möglich, über `CODECOPY` bzw. `CALLDATACOPY` einen bestimmten Codeausschnitt bzw. Eingangsparameter, welche sich im Calldata-Bereich befinden, in das Memory zu schreiben. Selbiges ist für Blockinformationen möglich, um zum Beispiel den Zeitstempel des aktuellen Blocks zu verwenden (mithilfe von `TIMESTAMP`). `JUMP`, `JUMPI` und `JUMPDEST` sind für das Steuern des Kontrollflusses zuständig und `LOG0` bis `LOG4` zum Logging von Daten. Abschließend existieren System-Operationen zum Erstellen (`CREATE`) oder Zerstören (`SELFDESTRUCT`) von Contracts oder zum Senden von Nachrichten an andere Contracts (z.B. `CALL`).

Der Zustand der EVM während der Laufzeit wird durch das Tupel

```
(block_state, transaction, message, code, memory, stack, pc, gas)
```

beschrieben. Der Parameter `block_state` repräsentiert dabei den globalen Zustand, welcher alle Accounts und deren Besitztümer sowie Storages beinhaltet. Während der Ausführung wird in jeder Runde die aktuelle Anweisung bestimmt, indem das Byte an der Stelle des Programmzeigers `pc` im auszuführenden Quelltext `code` verwendet wird. Jede Operation hat ihre eigene Definition darüber, wie sie auf das Tupel anzuwenden ist. Die Instruktion `ADD` holt beispielsweise zwei Werte vom Stack, addiert diese und legt das Ergebnis zurück auf den Stack. Danach wird der Parameter `gas` um den Wert 1 reduziert und der Programmzeiger um denselben erhöht [20].

Da diese Bytecode-Sprache eine Low-Level-Programmiersprache darstellt, welche sehr effizient für Maschinen, aber nur sehr aufwändig für Menschen lesbar ist, wurden höhere Sprachen entwickelt, welche für die Programmierung besser geeignet sind. Der Code dieser Programmiersprachen wird mithilfe eines Compilers in EVM-Bytecode übersetzt, damit dieser von der EVM ausgeführt werden kann. Näheres zu höheren Sprachen befindet sich im nächsten Abschnitt.

2.3.5 Solidity

Es existieren einige Sprachen, mit denen Smart Contracts für die EVM implementiert werden können. Neben den Programmiersprachen *LLL* [8], *Serpent* [13], *Mutan* [9] und *Viper* [15] ist die am meisten verbreitete und für diese Diplomarbeit relevante Sprache *Solidity* [27]. Ein Beispiel für einen Solidity-Contract findet sich in Abbildung 2.1.

Bei Solidity handelt es sich um eine High-Level-Programmiersprache, welche zwar auf *JavaScript* basiert, im Gegensatz dazu allerdings statisch typisiert ist. Die Sprache

```
pragma solidity >=0.4.0 <0.7.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Abbildung 2.1: Beispiel für einen Smart Contract [27]

ist Turing-vollständig und verfügt über Vererbung, Polymorphie, Bibliotheken und benutzerdefinierte Typen [52]. Der Code kann beispielsweise in der online verfügbaren Entwicklungsumgebung *Remix* [11] [12] geschrieben, dort über den Solidity-Compiler in EVM-Bytecode übersetzt und anschließend zum Beispiel mithilfe von *Go Ethereum* [7] in die Blockchain hochgeladen werden.

Die Hauptstruktur in Solidity ist ein `contract`, welcher über das gleichnamige Schlüsselwort gekennzeichnet wird. Dabei besteht eine strukturelle Ähnlichkeit zu Klassen in objektorientieren Sprachen. Ein Contract besteht aus Variablen, die im Storage gespeichert werden, einem Konstruktor und Funktionen, welche auf die Storage-Variablen zugreifen können. Der Konstruktor einer Contract-Instanz kann nur einmal und zwar während der Erzeugung der jeweiligen Instanz aufgerufen werden und ist bspw. für die Initialisierung der Storage-Variablen zuständig. Seit der Solidity-Version 0.4.22 werden Konstruktoren mit dem Schlüsselwort `constructor` gekennzeichnet.

Funktionen sind mit `function` markiert, werden über eine Signatur, bestehend aus Funktionsnamen und Eingangsparametertypen, identifiziert und führen eine bestimmte Funktionalität aus, womit auch die Rückgabe eines Wertes möglich ist. Unterschieden wird zwischen öffentlichen (`public`) und privaten (`private`) bzw. internen (`internal`) und externen (`external`) Funktionen. Ferner existieren solche, die nur dazu da sind, Werte aus dem Storage abzufragen, ohne diesen zu verändern (gekennzeichnet mit `view`) bzw. Werte zu retournieren, ohne überhaupt auf diesen zuzugreifen (mit `pure` deklariert). Eine Funktion kann außerdem mit `payable` gekennzeichnet werden. Über dieses Schlüsselwort kann der Contract ETH empfangen, also seinen Besitz erhöhen.

Des Weiteren verfügt Solidity über übliche Kontrollstrukturen wie bedingte Anweisungen und Schleifen. Die meisten in JavaScript verfügbaren Strukturen sind auch in Solidity möglich – mit Ausnahme von `goto` und `switch`. Eine Liste der Solidity-Kontrollstrukturen ist in Tabelle 2.2 zu finden.

Weitere sprachspezifische Eigenschaften sind `modifier` bzw. `require`-Statements.

`if`
`else`
`while`
`do`
`for`
`break`
`continue`
`return`

Tabelle 2.2: Liste der Kontrollstrukturen in Solidity

Diese geben an, dass beim Aufruf einer Funktion eine bestimmte Eigenschaft erfüllt sein muss, andernfalls wird der Funktionsaufruf abgebrochen und das verbleibende Gas an den Aufrufer retourniert, ohne Zustandsänderungen zu übernehmen. Im EVM-Bytecode wird dafür ein `REVERT` durchgeführt. Im Gegensatz dazu wird beim Fehlschlag einer `assert`-Instruktion der Aufruf abgebrochen, ohne das verbleibende Gas zurückzugeben, was der EVM-Anweisung `INVALID` entspricht.

Eine weitere Besonderheit von Solidity sind *Events*. Ein `event` ermöglicht die Verwendung von Logging in der EVM, welches mithilfe der Opcodes `LOG0` bis `LOG4` durchgeführt wird. Dies lässt außerdem die Anwendung von JavaScript-Callbacks in der Benutzeroberfläche von DApps zu, um auf bestimmte Events zu reagieren. Wie auch Funktionen werden Events über eine Signatur, bestehend aus Eventname und Eingangsparametertypen, identifiziert.

Bereits erwähnt wurde die statische Typisierung von Solidity. Ein Auszug aus den möglichen Typen befindet sich in Tabelle 2.3. Zusätzlich ist die Definition von eigenen Typen möglich. Zu diesem Zweck gibt es Mappings, Strukturen und Enumerationen, welche mit den Schlüsselwörtern `mapping`, `struct` bzw. `enum` erstellt werden können.

Des Weiteren definiert jeder Contract selbst einen eigenen Typ. Diese können explizit vom und zum Typ `address` konvertiert werden. In einem Contract kann zudem ein neuer Contract instanziiert werden, damit dieser in weiterer Folge aufgerufen werden kann. Das wird in Solidity mittels `new ContractName(...)` bewerkstelligt und auf den EVM-Befehl `CREATE` übersetzt.

Mithilfe von speziellen globalen Variablen kann auf Nachrichten, Transaktionen bzw. Blockchain-Daten zugegriffen werden. Die Variable `msg.sender` ermöglicht beispielsweise den Zugriff auf den Sender, `msg.value` auf den Wert einer Nachricht. Über `block.timestamp` kann der Zeitstempel des aktuellen Blocks, über `tx.origin` der Sender der aktuellen Transaktion ausgelesen werden. Diese Variablen korrespondieren also mit den entsprechenden EVM-Opcodes.

Weitere Besonderheiten sind beispielsweise die Features `call`, `delegatecall` sowie `selfdestruct`, welche in die gleichnamigen EVM-Befehle übersetzt werden. Diese und weitere Funktionalitäten von Solidity spielen in der aktuellen Diplomarbeit nur eine

Typ	Beschreibung
`bool`	Repräsentiert die Werte `true` und `false`.
`int`	Repräsentiert vorzeichenbehaftete ganze Zahlen. Typen existieren in Schritten von 8 (`int8` bis `int256`).
`uint`	Repräsentiert vorzeichenlose ganze Zahlen. Typen existieren in Schritten von 8 (`uint8` bis `uint256`).
`fixed`	Repräsentiert vorzeichenbehaftete Festkommazahlen in variablen Größen mithilfe von `fixedMxN`.
`ufixed`	Repräsentiert vorzeichenlose Festkommazahlen in variablen Größen mithilfe von `ufixedMxN`.
`address`	Repräsentiert eine Ethereum-Adresse in der Größe von 20 Bytes.
`byte`	Repräsentiert ein Byte-Array fixer Größe. Mögliche Typen sind `bytes1` bis `bytes32`.
`bytes`	Repräsentiert ein Byte-Array dynamischer Größe.
`string`	Repräsentiert eine UTF-8-codierte Zeichenfolge dynamischer Größe.

Tabelle 2.3: Auszug aus den Datentypen in Solidity [27]

untergeordnete Rolle und werden aus diesem Grund nicht im Detail beleuchtet. Weitere Informationen dazu sind in der Solidity-Dokumentation [27] enthalten.

KAPITEL 3

Dekompilierung

Beim Entwickeln von Software ist die übliche Vorgehensweise das Schreiben von Code in einer höheren Programmiersprache, die für den Menschen leicht verständlich, aber für die Maschine, auf welcher der Code letztendlich ausgeführt werden soll, nicht zu gebrauchen ist. *Forward Engineering Tools*, wie zum Beispiel der *Compiler*, sorgen dafür, dass der Code einer *High-Level Programming Language* (HLL) in einen *Bytecode*, also in einen für die Maschine ausführbaren Code, übersetzt wird.

Für die umgekehrte Variante, also um vom für den Menschen unlesbaren Maschinencode zum Code der höheren Programmiersprache zu gelangen, werden sogenannte *Reverse Engineering Tools* benötigt. Die Anwendungsgebiete für diese Art der Softwareentwicklung sind beispielsweise die Bereiche Softwarewartung bzw. -sicherheit.

In diesem Kapitel wird auf die für diese Diplomarbeit relevanten Aspekte der *Dekompilierung* – das Hauptkonzept von *Reverse Engineering* – eingegangen. Wichtige und für die weiteren Kapitel relevante Begriffe und Methoden werden angeführt und näher erläutert.

3.1 Allgemein

Zwei relativ einfache Reverse-Engineering-Konzepte, welche die Untersuchung von binären Applikationen ermöglichen, sind *Disassembling* und *Debugging*. Ohne diese Hilfsmittel sind Binärcodes nur Sequenzen von hexadezimalen Zeichen, die für Menschen nur schwer lesbar sind. Diese Tools sind in der heutigen Zeit für die Analyse von Software jedoch nicht mehr ausreichend, da die Größe der Programme enorm gestiegen ist. Das Disassembling etwa, welches ein Mapping von Binärcodes zu Instruktions-Mnemonics durchführt und dadurch lediglich individuelle Anweisungen und keine strukturellen Informationen zur Verfügung stellt, macht die schnelle Analyse von großen Applikationen unmöglich. Da allerdings aufgrund der stark wachsenden Schadsoftware und der sich schnell ändernden

Angriffsmuster die Notwendigkeit besteht, die Applikationen umgehend zu analysieren, wurde das Konzept der Dekompilierung eingeführt [32].

Laut der Doktorarbeit von Cristina Cifuentes mit dem Titel *Reverse Compilation Techniques* [21] ist ein *Decompiler* ein Tool, welches ein Quellprogramm, zumeist in einer Maschinensprache geschrieben, in ein äquivalentes Zielprogramm einer HLL umwandelt. Diese Technik ist also die umgekehrte Variante zur *Kompilierung*, welche ein High-Level-Programm in ausführbaren Binärcode übersetzt. Dabei ist es nicht von Relevanz, welche Maschinen- bzw. höhere Programmiersprachen von der Übersetzung betroffen sind.

Eine interessante Tatsache ist, dass die Struktur eines Decompilers auf der eines Compilers basiert. Dieser äquivalente Aufbau beinhaltet Phasen, in denen das Programm, geschrieben in einer Quellsprache, von einer Repräsentation in die nächste konvertiert wird, bis die Applikation in einer Zielsprache verfügbar ist. Im Gegensatz zur Kompilierung bringt die Dekompilierung jedoch schwierige und teils gar unlösbare Herausforderungen mit sich. Dies ist den Tatsachen geschuldet, dass der Compiler ein Input-Programm voraussetzt, welches syntaktisch einer bekannten Zielsprache entspricht, dadurch wesentliche Informationen wie Funktionen, Variablen und Typen zur Verfügung stehen und der Output nicht sonderlich strukturiert sein muss, da er üblicherweise nur von einer Maschine interpretiert wird. Im Gegensatz dazu sind die (umgekehrten) Informationen, die ein Decompiler zur Verfügung hat, nicht besonders strukturiert oder gar verschleiert, das Wiederherstellen von gewissen Strukturen bzw. Informationen dadurch schwierig oder gar unmöglich und eine lesbare Ausgabe von hoher Relevanz [32].

In Abbildung 3.1 findet sich der Aufbau eines Decompilers nach Cifuentes [21]. Diese Modularisierung ist notwendig, damit bei Bedarf die Quell- oder die Zielsprache des zu übersetzenden Programms einfach ausgetauscht werden kann. Da nur das erste bzw. letzte Modul von den gewählten Sprachen abhängig und alle Phasen dazwischen hingegen generisch sind, ist es möglich die Maschine oder die HLL auszutauschen, indem lediglich das betroffene Modul angepasst wird. Die generischen Stages bleiben dadurch jedenfalls unberührt.

Im maschinenabhängigen *Frontend* wird der Bytecode nach den syntaktischen und semantischen Analysen zu Code einer Zwischendarstellung bzw. zu einem *Kontrollflussgraphen* (CFG) transferiert. Diese Artefakte benötigt der Decompiler für weitere Analysen.

Das Modul *Universal Decompiling Machine* repräsentiert die zentrale und unabhängige Stelle zur Durchführung der Analysen. Die darin durchgeführte Datenflussanalyse ist dazu da, die Zwischendarstellung des Codes zu verbessern, sodass in weiterer Folge Ausdrücke einer höheren Programmiersprache reproduziert werden können. Die Kontrollflussanalyse versucht anhand des CFGs die High-Level-Struktur jeder einzelnen Funktion wiederherzustellen. Solche Strukturen sind beispielsweise bedingte Anweisungen und Schleifen.

Das abschließende *Backend* ist abhängig von der betroffenen HLL und führt somit die eigentliche Codeerzeugung durch. Für die Erstellung von High-Level-Code der Zielsprache werden unter anderem die Ergebnisse der vorhergehenden Phasen herangezogen.

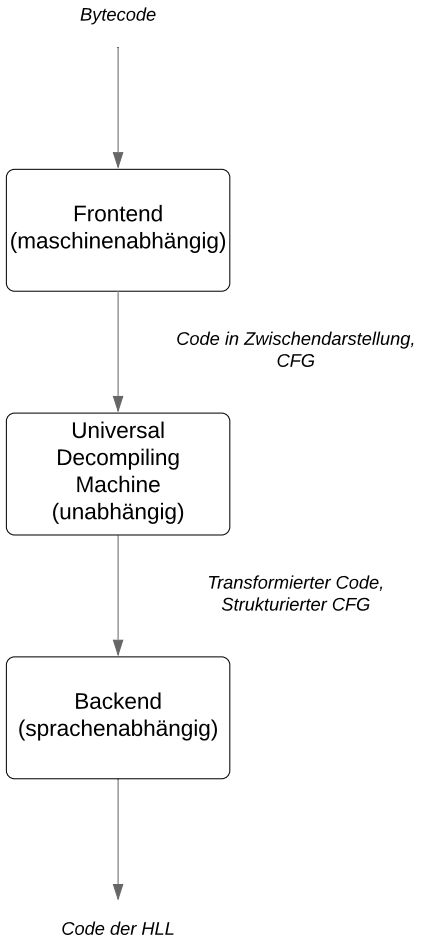

Abbildung 3.1: Die Module eines Decompilers nach Cifuentes [21]

In den weiteren Abschnitten dieses Kapitels werden die relevanten Konzepte genauer beschrieben.

3.2 Disassembler

Als Disassembler bezeichnet man einen Programmteil eines Decompilers, welcher den in einer Maschinensprache zugrunde liegenden Code in eine für Menschen besser lesbare Darstellung umwandelt. Im Konkreten wird also zum Beispiel Binär- bzw. Bytecode zum Quelltext einer Assembler-Sprache transferiert. Für diese Repräsentation werden Operationscodes (kurz: *Opcodes*), welche mithilfe von Zahlenwerten ausgedrückt werden, in *Mnemonics* entschlüsselt – es existiert also eine definierte Zuordnung von Opcodes auf mnemonische Kürzel, die für den Menschen verständlicher sind als Zahlenwerte [21].

Diese Darstellung wird verwendet, um Programme, von welchen der HLL-Code nicht

verfügbar ist, anhand des Maschinencodes z. B. auf Sicherheitslücken analysieren zu können. Diese Tatsache lässt die Verwendung eines Disassemblers als Debugging-Tool zu. Da ein Quellcode nach dem Disassembling zwar für den Menschen lesbarer ist, jedoch lediglich aus individuellen Anweisungen ohne jegliche Strukturen besteht, sind nach dieser Tätigkeit noch weitere Dekompilierungsschritte erforderlich, welche in den folgenden Abschnitten erläutert werden.

In Tabelle 3.1 ist ein Beispiel für ein Mapping von Opcodes zu Mnemonics angeführt. Beispielsweise wird der Opcode `01` in das Kürzel `ADD` übersetzt. Die Anweisung führt die Operation `a + b` aus, verbraucht dabei zwei Eingangs- und hat das Ergebnis der Addition als Ausgangsparameter.

Opcode	Mnemonic	Operation
00	STOP	Stoppt Programmausführung
01	ADD	a + b
02	MUL	a * b
03	SUB	a - b
04	DIV	a / b
05	SDIV	a / b
06	MOD	a % b
07	SMOD	a % b
08	ADDMOD	(a + b) % c
09	MULMOD	(a * b) % c
0A	EXP	a^b

Tabelle 3.1: Auszug aus dem Mapping von EVM-Opcodes zu Mnemonics [30]

3.3 Erstellung der Zwischendarstellung

Nach Cristina Cifuentes [21] ist das Frontend eines Decompilers unter anderem dazu da, den Quellcode der Maschinensprache in eine vorübergehende Repräsentation zu bringen. Diese Zwischendarstellung wird benötigt, damit in den weiteren Phasen des Decompilers die Daten- und Kontrollflussanalyse durchgeführt werden können. Anders als die spätere Ausgabe des Decompilers ist diese Zwischendarstellung unabhängig von der Zielsprache, damit diese mit wenig Aufwand ausgetauscht werden könnte.

Die Zwischendarstellung wird mithilfe von *Three-Address Code* repräsentiert. Dabei wird die stackbasierte Notation, die der Bytecode bzw. das Disassembling darstellt, in eine registerbasierte Form gebracht. Eine Anweisung in *Three-Address Code* entspricht der folgenden Struktur:

```
x := y op z
```

Die Kennzeichen `x`, `y` und `z` repräsentieren dabei die Adressen der betroffenen Register. Der logische oder arithmetische Operator `op` wird auf die Werte an den Adressen `y` und

z angewandt und das Ergebnis mithilfe des Zuweisungsoperators := in das Register mit der Adresse x geschrieben. Diese Darstellung ähnelt somit stark einem Ausdruck einer HLL.

3.4 Erstellung des Kontrollflussgraphen

Im maschinenabhängigen Frontend wird auf Grundlage des Binärcodes neben dem Code in Zwischendarstellung auch der *Control Flow Graph* (CFG) erstellt. Dabei wird der Bytecode nach dem Disassembling anhand gewisser Merkmale in kohärente Blöcke – die sogenannten *Basic Blocks* – eingeteilt, welche in Verbindung zueinander stehen und dadurch den Kontrollflussgraphen bilden.

3.4.1 Basic Blocks

Ein Programm besteht üblicherweise aus Anweisungen und Daten. Als *Basic Block* wird eine Sequenz von atomaren Anweisungen gesehen, welche den Kontrollfluss nicht ändert. Ein solcher Block kann mehrere Vorgänger- bzw. Nachfolger-Blöcke, jedoch nur exakt einen definierten Einstiegs- bzw. Ausstiegspunkt haben. Wird also eine Anweisung eines Basic Blocks ausgeführt, so werden auch alle anderen, in diesem Block enthaltenen Operationen ausgeführt. Anweisungen, die den Kontrollfluss ändern und somit das Ende eines Blocks markieren, wären beispielsweise Sprung-Instruktionen, Funktionsaufrufe oder Programm-Terminierungen. Unter Anwendung dieser Technik kann ein Programm eindeutig in eine Menge von sich nicht überlappende, kohärente Basic Blocks aufgeteilt werden, welche aufgrund eines definierten Kontrollflusses in Verbindung zueinander stehen [21].

3.4.2 Control Flow Graph

Nach Cifuentes [21] ist ein Kontrollflussgraph ein gerichteter Graph, welcher, wie der Name bereits erahnen lässt, den Kontrollfluss eines Programms repräsentiert. Ein Beispiel dazu ist in Abbildung 3.2 visualisiert. Die Knoten eines solchen Graphen sind die Basic Blocks, welche wie beschrieben die Instruktionen des Programms beinhalten. Diese Knoten sind mit Kanten verbunden, die den Kontrollfluss kennzeichnen. Für die Erstellung des CFGs werden die Basic Blocks abhängig von ihrer letzten Instruktion in die folgende Klassifizierung eingeteilt:

1. *1-Way Basic Block*: Der Block endet mit einem nicht bedingten Sprung und hat folglich nur eine Ausgangskante.
2. *2-Way Basic Block*: Der Block endet mit einem bedingten Sprung und hat folglich zwei Ausgangskanten.
3. *n-Way Basic Block*: Der Block endet mit einem indizerten Sprung. Abhängig von der Anzahl der möglichen Sprungadressen entstehen n Ausgangskanten.

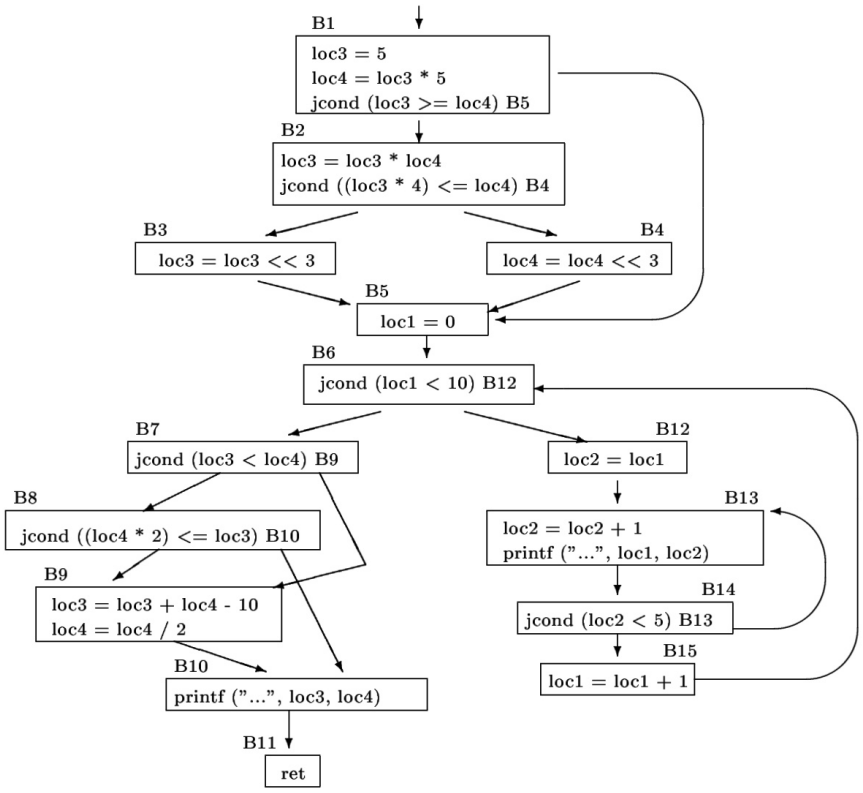

Abbildung 3.2: Beispiel eines CFGs aus dem Paper *Structuring Decompiled Graphs* von Cristina Cifuentes [22]

4. *Call Basic Block*: Der Block endet mit einem Funktionsaufruf.

5. *Return Basic Block*: Die letzte Instruktion des Blocks stellt eine Terminierung des Programms dar. Es existieren daher keine Ausgangskanten.

6. *Fall Basic Block*: Der Kontrollfluss läuft in die Instruktion mit der nächsten Programmadresse über. Dieser Knoten wird als *Fallthrough* bezeichnet und hat exakt eine Ausgangskante.

Die Wiederherstellung des CFGs auf Basis des Disassemblings erfolgt unter Zuhilfenahme eines *CFG Recovery Algorithms*. Dabei werden alle Basic Blocks rekursiv analysiert und zu jedem potenzielle Nachfolger-Blöcke ermittelt. Die Kanten des CFGs werden also unter anderem bestimmt, indem ein Block mit einem Sprung als abschließende Operation und dessen Zielblock verknüpft werden. Die Reproduktion der Sprungadressen stellt allerdings nicht immer eine triviale Aufgabe dar. Ein *indirekter Sprung*, bei welchem das Sprungziel abhängig vom Vorgänger-Block ist, hat anders als ein *direkter Sprung* keine statischen Informationen über seine Nachfolger. Die Challenge besteht also darin, so viele indirekte Sprünge wie möglich aufzulösen, damit ein vollständiger CFG reproduziert

werden kann, indem beispielsweise ein dynamischer Durchlauf des Graphen anhand der einzelnen Programmfunktionen durchgeführt wird [47].

Eine weitere Herausforderung der CFG-Erstellung ist, dass der Graph nach dem Durchlaufen des Algorithmus häufig aufgebläht, also mit Kanten versehen ist, die laut dem ursprünglichen Programm nicht eintreten können. Mithilfe von Graph-Optimierungen ist die Entfernung von unnötigen oder redundanten Sprüngen möglich. Die angewandte Methode zur Optimierung des CFGs wird *Peephole Optimization* genannt. Diese Technik untersucht kleine Sequenzen von Instruktionen und ersetzt diese durch noch kleinere oder effizientere Abfolgen [21].

3.5 Datenflussanalyse

Unter *Datenflussanalyse* versteht man alle Tätigkeiten, die auf Grundlage des Flusses der Daten entlang aller möglichen Ausführungspfade des Programms durchgeführt werden, um die in Abschnitt 3.3 erzeugte Zwischendarstellung des Codes zu optimieren. Dabei ist in erster Linie die Gewinnung von Informationen notwendig, die in weiterer Folge für die Optimierungen benötigt werden. Zu den für diese Arbeit wichtigsten Tätigkeiten gehört jedenfalls die Verbesserung der Darstellung von Variablen.

Live Variable Analysis oder *Liveness Analysis* ist eine Datenflussanalyse, die für jeden möglichen Punkt im Programm berechnet, welche Variablen an ebendiesem aktiv sind. Eine Variable ist *aktiv* oder auch *live* an einem bestimmten Punkt, wenn sie einen Wert beinhaltet, der in der weiteren Programmausführung noch benötigt wird. Ein Zugriff auf eine neu definierte Variable, bevor diese abermals beschrieben wird, würde diese Bedingung somit erfüllen. Anderenfalls wird die Variable *inaktiv* oder *dead* genannt, womit sie mithilfe von weiteren Optimierungen im Rahmen der Datenflussanalyse entfernt werden könnte [16].

Zu diesen Tätigkeiten zählt zum Beispiel die sogenannte *Dead Code Elimination*. Unter Zuhilfenahme der Informationen, die aus der Liveness-Analyse gewonnen werden, kann eine Anweisung, welche eine inaktive Variable erzeugt, gänzlich aus dem Code entfernt werden [21].

Eine weitere Optimierung ist *Constant Folding*. Dieses Konzept beschreibt die Simplifikation von Operator-Ausdrücken mit bekannten konstanten Werten als Operanden. Beispielsweise kann der Ausdruck `3 + (6 * (5 - 2)) / 2` unter Anwendung von arithmetischen Regeln einfachheitshalber zum konstanten Wert `12` transferiert werden. Diese Optimierung steht in enger Verbundenheit mit *Constant Propagation*, damit eine Variable durch ihren konstanten Wert ersetzt werden kann [44].

Constant Propagation bedeutet also, die Zuweisung eines konstanten Wertes zu einer Variable auf alle möglichen Vorkommnisse dieser Variable zu übertragen, damit diese in weiterer Folge gänzlich aus dem Programm entfernt werden kann. Zum Beispiel kann der Ausdruck `a := 3; b := a` mit `a := 3; b := 3` ersetzt werden. Für die Entfernung

des Ausdrucks `a := 3`, sofern der Wert der Variable `a` im weiteren Programmverlauf nicht mehr benötigt wird, ist die erwähnte *Dead Code Elimination* zuständig [44].

Copy Propagation ist ein ähnliches Konzept. Im Unterschied zu *Constant Propagation* werden hier jedoch keine konstanten Werte, sondern Zuweisungen von Variablen propagiert. Beim Codestück `a := b; c := 3 + a` ist es bspw. möglich, die Variable `a` zu entfernen und im zweiten Statement durch die Zuweisung der Variable `b` zu ersetzen, sodass die Anweisung `c = 3 + b` resultiert. Für die Eliminierung der Variable `a` ist klarerweise wieder *Dead Code Elimination* zuständig [21].

Man kann leicht erkennen, dass die beschriebenen Konzepte eng miteinander verbunden sind, was den Übergang von einem Konzept zum anderen fließend erscheinen lässt.

3.6 Kontrollflussanalyse

Der CFG, welcher mithilfe der Techniken aus Abschnitt 3.4 erstellt wurde, setzt sich aus miteinander verbundenen Blöcken zusammen und repräsentiert somit einen Kontrollfluss, der ohne weitere Analyse jedoch nicht auf die Kontrollstrukturen der HLL schließen lässt. Für die Wiederherstellung von Konstrukten wie Schleifen oder bedingten Anweisungen bedarf es somit der Anwendung einer *Kontrollflussanalyse*.

Die *Kontrollflussanalyse* reproduziert also Kontrollstrukturen der HLL auf Basis des CFGs. Um diese Erkennung durchzuführen, wird beispielsweise die Technik *Structural Analysis* angewandt, also eine strukturelle Analyse durchgeführt. In dieser Phase wird der Graph mithilfe von *Pattern Matching* auf bekannte Konstrukte durchsucht und die Kontrollstrukturen als Baum dargestellt. Dieser wird anschließend für die Erzeugung des High-Level-Codes verwendet [32].

Der Algorithmus, der für die strukturelle Analyse verwendet wird, führt ein Matching von bekannten Schemata bzw. Patterns auf lokale Regionen des CFGs durch, indem wiederholend die Knoten in umgekehrter Reihenfolge durchlaufen werden. Diese Schemata bzw. Patterns sind vordefiniert und jedes einzelne beschreibt dabei die Form einer High-Level-Kontrollstruktur, die es zu erkennen gilt. Wurde ein Match gefunden, wird die korrespondierende Region zu einem einzigen Knoten reduziert, welcher die Information über die Kontrollstruktur beinhaltet. Abbildung 3.3 zeigt diesen Prozess an einem CFG. Im ursprünglichen, linken Graph sind alle Knoten separat dargestellt. Nach dem ersten Durchlauf des Algorithmus werden die oberen drei Knoten als `if-then-else`-Konstrukt erkannt und zu einem Knoten zusammengeklappt. Der resultierende Graph wird weiter zu einem `do-while`-Konstrukt reduziert. Schlussendlich würde daraus folgender Code entstehen:

```
do { if (c1) then {...} else {...} } while (c2)
```

Können vom Algorithmus keine weiteren Strukturen detektiert werden, wurden entweder alle ursprünglichen Kontrollstrukturen korrekt wiederhergestellt (*proper region*) oder es verbleiben Sprünge (z. B. `goto`-Anweisungen) im dekompilierten Code, damit der

Kontrollfluss erhalten bleibt (*improper region*). Das Ziel von heutigen Decompilern ist es, so viele Kontrollstrukturen wie möglich wiederherzustellen und mit möglichst wenigen Sprüngen auszukommen [19].

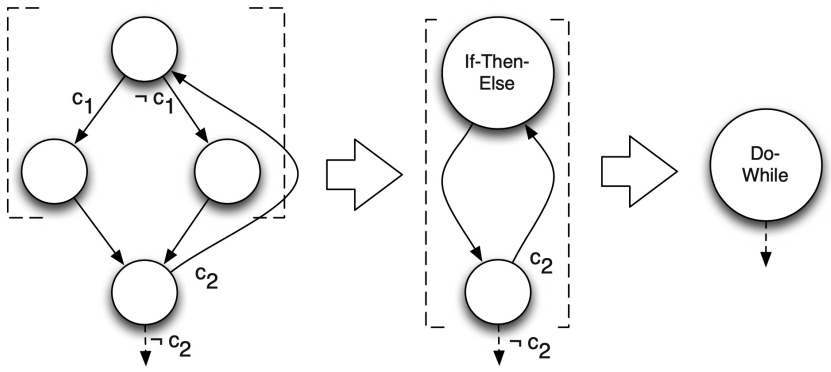

Abbildung 3.3: Beispiel einer strukturellen Analyse nach Schwartz et al. [19]

3.6.1 Strukturierung von Schleifen

Die möglichen Arten von strukturierten Schleifen sind in Abbildung 3.4 dargestellt. *Pre-tested loops*, bei welchen die Bedingung zu Beginn der Schleife evaluiert wird, sind beispielsweise die Konstrukte `while` oder `for`. Der *Post-tested loop* entspricht einer `do-while`-Schleife, bei welcher die Schleifenbedingung bekanntlich am Ende geprüft wird. Beim *Infinite loop* gibt es keine Evaluierung der Schleifenbedingung, da diese Art unendlich oft ausgeführt wird. Ein *Multiexit loop* entspricht einer Schleife mit beliebig vielen Austrittspunkten, das sind z. B. `break`-Anweisungen [21].

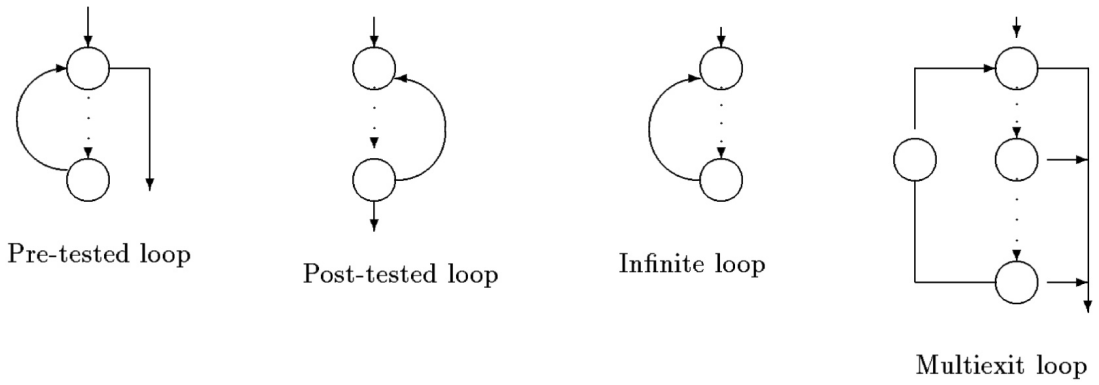

Abbildung 3.4: Patterns für die Strukturierung von Schleifen nach Cifuentes [21]

Domination

Um diese Patterns im CFG erkennen zu können, müssen spezielle Techniken angewandt werden. Ein solches Schlüsselkonzept in der Kontrollflussanalyse ist *Domination*. Der Dominator-Knoten `d` dominiert dabei den Knoten `n`, wenn jeder Pfad im Graph `G` vom Startknoten `n0` zum Knoten `n` auch den Dominator `d` inkludiert. Der Knoten `n` kann also nicht durchlaufen werden, ohne zuvor den Dominator `d` passiert zu haben. Die Relation zwischen den beiden Knoten wird mit `d` **`dom`** `n` ausgedrückt [19].

Back Edge

Anhand dieses Dominator-Prinzips können Schleifen erkannt und in weiterer Folge als solche dargestellt werden. Eine natürliche Schleife wird dabei anhand einer sogenannten *Back Edge* identifiziert. Eine gerichtete Kante `(s,d)` definiert eine *Back Edge*, wenn die Relation `d` **`dom`** `s` gilt. Das heißt, es existiert ein Schleifenknoten `s`, der von seinem Nachfolger `d` dominiert wird. Der Knoten `d` muss aufgrund der Domination auch ein Vorgänger von `s` sein, womit `d` den Beginn einer Schleife repräsentieren muss. Die Schleifenknoten der *Back Edge* `(s,d)` sind alle Knoten, die *Tail* `s` erreichen können, ohne *Header* `d` durchlaufen zu müssen – also alle Knoten, die zwischen `d` und `s` liegen, sowie Header `d` und Tail `s` selbst [2] [19].

Loop Successor Refinement

Ein weiteres Konzept, welches bei der Erkennung von Schleifen im Rahmen der Dekompilierung Anwendung findet, nennt sich *Loop Successor Refinement*. Unter Zuhilfenahme dieser Technik ist es möglich, alle vorzeitigen Schleifenaustritte (entsprechen `break`-Anweisungen) zu bestimmen, damit diese als Schleifenknoten registriert werden. Dies ist notwendig, damit die zugehörige Region der Schleife am Ende der Restrukturierung exakt einen Ausstiegspunkt besitzt, zu dem alle ursprünglichen vorzeitigen Austritte verweisen. Erst dann kann eine Schleife mit mehreren Ausstiegen zuverlässig als solche erkannt werden [54].

3.6.2 Strukturierung von Bedingten Anweisungen

Die für diese Arbeit relevanten Patterns für die Strukturierung von bedingten Anweisungen befinden sich in Abbildung 3.5. Das linke Schema *Single branch conditional* entspricht einer `if-then`-Anweisung ohne `else`-Zweig. Dabei wird im obersten Knoten eine bestimmte Bedingung überprüft (`if`) und einer der Nachfolger nur ausgeführt, wenn die Überprüfung erfolgreich verläuft (`then`). Unabhängig vom Ausgang des Checks wird danach in jedem Fall der unterste Knoten ausgeführt.

Die *Conditional*-Struktur auf der rechten Seite der Abbildung entspricht einer `if-then-else`-Anweisung. Im Gegensatz zur `if-then`-Instruktion gibt es einen zweiten Zweig, der nur unter einer bestimmten Bedingung ausgeführt wird und zwar falls die Überprüfung nicht erfolgreich verläuft. Dieser Knoten kennzeichnet den `else`-Block. Nach der

Ausführung von genau einem der beiden Zweige führt der Kontrollfluss jedenfalls wieder zusammen, womit das Ende der bedingten Anweisung erreicht ist.

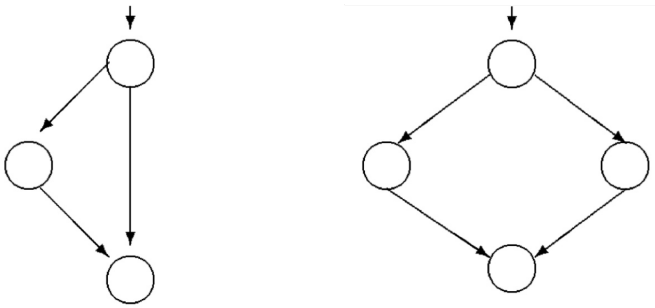

Abbildung 3.5: Auszug aus den Patterns für bedingte Anweisungen nach Cifuentes [21]

3.7 Code-Generierung

Die *Universal Decompiling Machine*, welche die Daten- und Kontrollflussanalyse durchführt, liefert als Output einen transformierten Zwischencode, welcher dem Code einer generischen HLL ähnelt und einen strukturierten CFG, also einen Graphen, der mit High-Level-Kontrollstrukturen angereichert wurde. Die Aufgabe des sprachabhängigen *Backends* ist es, aus diesen beiden generischen Artefakten HLL-Code der Zielsprache zu generieren. Diese Tätigkeit übernimmt der sogenannte *Code Generator*. Die Herausforderung dabei ist, für jeden Ausdruck bzw. jede Kontrollstruktur der Zwischendarstellung, sei es eine Zuweisung, ein Vergleich, eine bedingte Anweisung, eine Schleife oder aber ein verbliebener Sprung, einen korrespondierenden Ausdruck in der Zielsprache parat zu haben [21].

KAPITEL 4

Related Work

In diesem Abschnitt werden in erster Linie die untersuchten und evaluierten Forschungsarbeiten, welche sich bereits ansatzweise mit der Dekompilierung von EVM-Bytecode beschäftigen, angeführt und beschrieben. Dadurch werden auch Tools relevant, die nicht notwendigerweise *Reverse Engineering* und die damit verbundene Code-Wiederherstellung betreiben, sondern auch solche, die zumindest ein *Disassembling* des rohen EVM-Bytecodes und die Erstellung eines *Kontrollflussgraphen* durchführen. Diese Tatsache inkludiert somit auch jene Arbeiten, die sich mit der Analyse von Smart Contracts hinsichtlich Sicherheitslücken beschäftigen. Ausgehend vom Paper *A Survey of Tools for Analyzing Ethereum Smart Contracts* [26], welches von di Angelo und Salzer verfasst wurde, wurden also alle Tools berücksichtigt, die sich entweder mit Reverse Engineering von EVM-Bytecode befassen (*Erays*, *Porosity*, *DSol*) oder zumindest einen CFG erstellen und gegebenenfalls anzeigen (*Vandal*, *Oyente*, *Mythril*, *teEther*). Jedes dieser Programme ist im Rahmen einer Forschungsarbeit entstanden und zudem öffentlich auf *GitHub* [6] unter einer Open-Source-Lizenz zugänglich.

Abschließend fasst dieses Kapitel die wesentlichste Literatur hinsichtlich allgemeiner Dekompilierungsprogramme zusammen. Die betroffenen Werkzeuge sind *Hex-Rays*, der *De-facto-Standard* in der Industrie, sowie die fortgeschritteneren Decompiler *Phoenix* und *Dream*, welche allesamt u. a. Code der Maschinensprache *x86* auf die höhere Programmiersprache *C* übersetzen. Auch in diesen Forschungsarbeiten finden sich relevante Konzepte der Dekompilierung, wie zum Beispiel die Technik *Structural Analysis*.

4.1 Oyente

Das Tool *Oyente*, welches im Artikel *Making Smart Contracts Smarter* [39] vorgestellt und von *melonproject* auf GitHub veröffentlicht wurde [40], stellt den Pionier für Analysen von Ethereum Smart Contracts dar. So wurde es bereits 2016, also im Jahr nach der Veröffentlichung von Ethereum, publiziert. Da es eines der ersten Tools war, die sich der

Analyse von Smart Contracts, sowohl auf Solidity- als auch auf EVM-Bytecode-Ebene widmeten, ist es ein Paradebeispiel für verwandte Arbeiten einer Forschung, die sich mit der Dekompilierung von EVM-Bytecode befasst. Diese Annahme wird auch von der Tatsache untermauert, dass einige weitere, in dieser Diplomarbeit berücksichtigte Schriftstücke, das Projekt Oyente referenzieren. Generell geht es in dieser Arbeit darum, mithilfe von symbolischer Ausführung Sicherheitslücken in Ethereum Smart Contracts zu detektieren.

Wie für die Dekompilierung vorausgesetzt führt Oyente ein Disassembling des EVM-Bytecodes durch, unterteilt die ermittelten Befehle anhand von bestimmten Opcodes in Basic Blocks und generiert daraufhin einen CFG. Dabei werden klare Sprünge, also solche, die ein statisch erkennbares Sprungziel haben bzw. *Fallthroughs* bei indirekten Sprüngen sofort bestimmt. Bei der symbolischen Ausführung des Codes wird versucht die verbleibenden, statisch nicht erkennbaren Sprungadressen wiederherzustellen. Aufbauend darauf überprüft das Tool den Smart Contract auf bestimmte Sicherheitslücken (wie z.B. *Integer Underflow*, *Reentrancy*), um zu verhindern, dass der Programmierer oder Verwender eines Contracts wegen bekannter Bugs Ether verliert. Ferner ist für die aktuelle Arbeit relevant, dass Oyente öffentliche Funktionen erkennen kann.

4.2 Mythril

Auch *Mythril* [25] untersucht Ethereum Smart Contracts anhand von symbolischer Ausführung auf Sicherheitslücken. Im zugehörigen Paper [41] wird in der Einleitung erwähnt, dass das Ethereum-Netzwerk damit an Sicherheit gewinnen soll, weshalb einige bekannte Security-Bugs, wie zum Beispiel *Reentrancy*, auch im Schriftstück angeführt werden.

Dabei beginnt Mythril ähnlich Oyente mit einem Disassembling des EVM-Bytecodes. Auf Grundlage dieser Tätigkeit wird daraufhin der Bytecode symbolisch ausgeführt, wodurch Sprünge dynamisch erkannt werden können. Anhand dieser symbolischen Ausführung wird anschließend der CFG erstellt, welcher im Gegensatz zu Oyente auch visualisiert werden kann.

Eine weitere Relevanz bezüglich des Themas dieser Diplomarbeit *Decompilation of EVM Bytecode* ist, dass Mythril externe Funktionen erkennen und unter Umständen auch die Signatur davon wiederherstellen kann. Außerdem wird im Paper von Bernhard Müller erwähnt, dass die Solidity-Anweisung `assert` im EVM-Bytecode erkennbar ist, da die gescheiterte Prüfung mit dem Opcode `INVALID` terminiert.

4.3 Porosity

Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode lautet der Name des Papers von Matt Suiche [49]. Dieses Schriftstück beschreibt ein Tool, welches EVM-

Bytecode zu Solidity-Code dekompiliert und gleichzeitig bestimmte Sicherheitslücken, wie zum Beispiel den *Reentrancy-Bug*, erkennt.

Dabei wird im Paper hauptsächlich auf die technischen Eigenschaften der verschiedenen EVM-Operationscodes eingegangen. Beispielsweise wird das Lesen bzw. Schreiben in den Storage mit den EVM-Instruktionen `SLOAD` bzw. `SSTORE` durchgeführt, die Zugriffsoperationen auf den flüchtigen Memory-Speicher sind hingegen `MLOAD` und `MSTORE`.

Porosity geht weiters davon aus, dass beinahe alle Basic Blocks mit der Anweisung `JUMPDEST` starten. Die dynamische Ausführung des Bytecodes ist notwendig, um die Stack-Werte bei jedem Sprung zu erkennen, damit die Sprungziele ausfindig gemacht werden können. Außerdem identifiziert das Tool Funktionen, die vom Entwickler des Contracts im Solidity-Code implementiert wurden, anhand der Instruktion `CALLDATALOAD`.

Für die Typerkennung von Adressen kann das direkte Schreiben der nicht optimierten 160-Bit-Adressmaske `0xffffffffffffffffffffffffffffffffffffffff` auf den Stack herangezogen werden. Außerdem ist der EVM-Opcode `CALLER`, welcher im Solidity-Code dem Ausdruck `msg.sender` entspricht, ein Anzeichen für eine Adresse, also den Datentyp `address`.

Ferner führt das Schriftstück an, dass beim EVM-Bytecode eines Smart Contracts zwei Teile unterschieden werden können. Der gesamte Bytecode beinhaltet den sogenannten *Pre-Loader*, also den Teil, der den *Runtime-Code* des Contracts in das Memory der EVM mithilfe der Anweisung `CODECOPY` kopiert. Der zweite Teil, also der Runtime-Code selbst, startet mit einem *Runtime Dispatcher*, der die Ausführung des Bytecodes zur Position der auszuführenden Funktion leitet. Mithilfe der ersten vier Bytes des SHA3-Hashes der Funktionssignatur weiß der Dispatcher, welcher Abschnitt des EVM-Bytecodes ausgeführt werden muss, also zu welcher Adresse im Code er springen muss. In den weiteren Abschnitten dieser Arbeit wird der *Runtime Dispatcher* gemäß anderer Referenzen mit *Function Selector* bezeichnet. Die ursprünglichen Funktionssignaturen können nur dann korrekt wiederhergestellt und dem korrespondierenden Codestück zugewiesen werden, wenn bei der Ausführung von Porosity das *Application Binary Interface* (ABI) des Contracts mitgeliefert wird. Details dazu siehe Abschnitt 5.1.

4.4 Erays

Das Programm *Erays* [51] beschäftigt sich mit Reverse Engineering von Ethereum Smart Contracts. Die Autoren des Papers *Erays: Reverse Engineering Ethereum's Opaque Smart Contracts* [55] beschreiben ein Tool, das die undurchsichtigen EVM-Contracts teilweise zu verständlicherem Code wiederherstellen kann. In diesem Artikel wird darauf hingewiesen, dass zum Zeitpunkt des Verfassens eine Million Contracts auf der Ethereum Blockchain existierten. Allerdings sind die meisten dieser Contracts ident, wodurch ca. 34.000 eindeutige Programme verbleiben. Davon sind nur 23 % transparent, z. B. über *Verified Contracts* [29], womit 77 % an Contracts verbleiben, von denen der Quellcode nicht bekannt ist.

Das Programm beginnt wie alle anderen mit einem Disassembling und der Identifizierung von Basic Blocks anhand bestimmter Aufteilungsopcodes. Die Operation `JUMPDEST`, welche die Zieladresse eines Sprungs darstellt, markiert dabei den Beginn, die Sprung-Opcodes `JUMP` bzw. `JUMPI` und die Terminierungsopcodes (Tabelle 2.1) das Ende eines Blocks. Nach diesen Tätigkeiten findet die Wiederherstellung des CFGs statt. Mithilfe folgender Regeln werden zu jedem Block dessen Nachfolger bestimmt:

1. Die Anweisung verändert den Kontrollfluss nicht und der Block läuft in seinen Nachfolger, also in den Block mit der nächsten Programm-Adresse, über.

2. Die Anweisung beendet die Ausführung. Ein Block welcher mit `STOP`, `REVERT`, `INVALID`, `RETURN` oder `SELFDESTRUCT` endet, hat somit keine Nachfolger.

3. Die Anweisung springt zu einem Block mit einer bestimmten Adresse. `JUMP` springt immer, `JUMPI` führt einen Sprung durch, wenn eine bestimmte Bedingung zutrifft bzw. läuft ansonsten in den Block mit der nächsten Adresse über.

Ein direkter Sprung ist relativ einfach zu bestimmen, indem die Zieladresse, welche im selben Block in dem sich der Sprung befindet, auf den Stack gelegt wurde, vom Stack geholt wird. Indirekte Sprünge sind hingegen komplexer. Bei diesen befindet sich das Sprungziel bereits beim Eintreten in den jeweiligen Block am Stack. Die Adresse wird also bereits in einem Vorgänger-Block auf den Stack gespeichert und kann somit je nach Vorgänger unterschiedlich sein. Da es in solchen Fällen nicht ausreichend ist, den Block, in welchem der Sprung stattfindet, separat zu betrachten, muss ein *CFG-Recovery-Algorithmus* angewandt werden, mit welchem ein vollständiger Durchlauf des CFGs anhand der tatsächlichen Stack-Werte emuliert wird.

Der dritte Dekompilierungsschritt wird *Lifting* genannt. In dieser Phase werden stackbasierte zu registerbasierten Anweisungen umgeschrieben, indem jedes der 1.024 in der EVM möglichen Stack-Worte auf ein eigenes Register abgebildet wird. Zusätzlich zu EVM-Instruktionen werden Befehle für die Ausführung von privaten Funktionen, `ASSERT` für die Sicherheitschecks des Programmierers bzw. Solidity-Compilers, zusätzliche Vergleichsoperatoren und die Operation `MOVE`, die aus den EVM-Opcodes `SWAP`, `DUP` bzw. `PUSH` abgeleitet wird, eingeführt.

In der Phase *Optimization* werden zahlreiche Compiler-Optimierungen angewendet. Die Zwischendarstellung des Codes wird mithilfe von Datenfluss-Optimierungen wie zum Beispiel „constant folding, constant propagation copy propagation and dead code elimination“ [55] verbessert. Dadurch wird der Code vereinfacht, weil etliche redundante Anweisungen entfernt werden, die nur für das Verschieben von Werten zwischen Registern zuständig sind.

Aggregation ist dafür zuständig, die Zwischenrepräsentation zu verbessern, indem Instruktionen mit analogen, aber kompakteren Versionen ersetzt werden – die sogenannten

Aggregated Expressions. In diesem Schritt werden aus Anweisungen also Ausdrücke, welche im Gegensatz zu ersteren beliebig geschachtelt werden können. In der von Erays verwendeten *Three-Address Form*, welche bereits in Abschnitt 3.3 präsentiert wurde, ist jeder Ausdruck eine Kombination aus einem *Write*-Register auf der linken und einem Operator sowie den *Read*-Registern auf der rechten Seite der Zuweisung. Das Register, in das geschrieben wird, repräsentiert zudem eine Variable, die mithilfe einer *Liveness Analysis* in den nachfolgenden Ausdrücken durch ihre Zuweisung ersetzt wird, solange diese Ersetzung valide ist. Mit dieser Technik werden nicht benötigte Variablen innerhalb eines Blocks entfernt.

Beim abschließenden Dekompilierungsschritt wird eine strukturelle Analyse durchgeführt, mit welcher versucht wird, High-Level-Kontrollstrukturen, wie zum Beispiel `while`-Schleifen und `if-then-else`-Ausdrücke, zu rekonstruieren. Dabei wird der CFG jeder externen Funktion durchlaufen und mithilfe von *Pattern Matching* versucht, den CFG zu reduzieren, also mehrere Blöcke in abstraktere Konstrukte zusammenzufassen.

Zudem wird in der Publikation festgehalten, dass Erays keinen vollständigen Decompiler darstellt, der kompilierbaren Solidity-Code erstellt. Dieses Tool hat also noch Verbesserungspotenzial hinsichtlich Lesbarkeit des Outputs, Variablen- bzw. Typerkennung (z. B. `mapping`), sowie strukturelle Analysen für die Identifikation von komplexeren Solidity-Kontrollstrukturen [55].

4.5 Vandal

Die Publikation *Vandal: A Scalable Security Analysis Framework for Smart Contracts* [18] beschreibt ein Tool namens *Vandal* [48], welches sich als Framework für Sicherheitsanalysen von Ethereum Smart Conrtracts anbietet.

Das *Vandal Framework* besteht dabei aus mehreren Komponenten. Der sogenannte *Scraper* extrahiert den EVM-Bytecode von Smart Contracts aus der Ethereum Blockchain. Daraufhin konvertiert der *Disassembler* den Bytecode zu menschenlesbaren, textuellen Operationscodes und ihren Argumenten, welche anhand von Programmadressen eindeutig identifiziert werden können.

In der nächsten Phase der Analyse-Pipeline übersetzt der *Decompiler* den stackbasierten Bytecode in eine registerbasierte Zwischenrepräsentation. Diese Darstellung legt Daten- und Kontrollfluss des Bytecodes offen. Mithilfe dieser Daten wird anhand einer symbolischen Ausführung inkrementell ein CFG erstellt. Diese CFG-Erstellung wird von den Autoren als größte Challenge interpretiert, da das Wiederherstellen der indirekten Sprünge keine triviale Tätigkeit darstellt. Auf Basis der zugrunde liegenden Artefakte werden daraufhin Kontroll- und Datenflussanalyse durchgeführt.

Der *Extractor* produziert abschließend die logischen Relationen, die die Semantik des Smart Contracts erfassen und von einem Sicherheitsanalyse-Tool gelesen werden können.

4.6 teEther

Das Paper *teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts* [37] führt ein weiteres Programm ein, welches sich mit der Analyse von EVM-Bytecode basierend auf bestimmten Sicherheitslücken beschäftigt. Im Gegensatz zu den bisher erwähnten Tools kann letzteres allerdings dazu verwendet werden, diese Schwachstellen zu erkennen und auszunutzen, um an das Kapital des fehlerhaften Contracts zu gelangen.

Technisch wird *teEther* [23] in mehrere Module unterteilt. Das CFG-Recovery-Modul führt ein Disassembling des EVM-Bytecodes durch und konstruiert daraufhin den CFG. In der nächsten Phase wird der Graph auf kritische und Status-verändernde Anweisungen durchsucht. Danach wird ein Pfad von der Wurzel des CFG bis zur kritischen Instruktion erstellt und basierend darauf mithilfe von symbolischer Ausführung *Path Constraints* erzeugt. Im letzten Modul wird anhand dieser Contraints ein *Exploit* zum Ausnutzen der Sicherheitslücken generiert.

Auch die Autoren Krupp und Rossow erwähnen die Problematik im Zusammenhang mit der vollständigen CFG-Erstellung, welche die Gegebenheit der indirekten Sprünge mit sich bringt. Dieses Thema wird adressiert, indem das Konzept namens *Backward Slicing* angewendet wird.

Ferner wird im Paper angeführt, dass bei der Modellierung der `SHA3`-Anweisung im Gegensatz zu anderen Instruktionen besondere Vorsicht geboten ist. Diese Operation bekommt als Argument eine Memory-Region und berechnet den *Keccak-256*-Hash des Inhalts. Der Solidity-Compiler verwendet diese Anweisung bspw. für die Datenstruktur `mapping`, welche einen *Key Value Store* darstellt. Der Zugriff auf einen Wert, welcher in einem `mapping` gespeichert ist, wird mit der Berechnung des *Keccak-256*-Hashes des Schlüssels und mit der Verwendung des Ergebnisses als Index für den Zugriff auf den Storage des Contracts durchgeführt.

4.7 DSol

In der Masterarbeit *Decompilation of Ethereum smart contracts* von Thomas Hybel aus dem Jahre 2018 [34] wurde ähnlich dieser Diplomarbeit die Dekompilierung von EVM-Bytecode zu Solidity-Code untersucht. Für diesen Zweck wurde das Tool *DSol* entwickelt, welches öffentlich auf GitHub unter einer MIT-Lizenz zugänglich ist [33].

DSol verwendet die Dekompilierungstechniken nach Cifuentes [21], um den Maschinencode zurück in High-Level-Code zu übersetzen. Dazu wird der EVM-Bytecode in eine Zwischenrepräsentation konvertiert, bevor Datenflussanalysen, welche das Abstraktionslevel des Contracts verbessern, durchgeführt werden. Außerdem wird anhand der wiederhergestellten Kontrollstrukturen eine abstrakte Syntax erzeugt, womit in weiterer Folge Code der höheren Programmiersprache Solidity reproduziert wird.

Als Einschränkungen des entwickelten Decompilers werden in der Arbeit unter anderem die Performance-Bottlenecks beim vollständigen Durchqueren des CFGs angeführt.

Des Weiteren besitzt das Tool keine Erkennung von Funktionssignaturen und keine Rekonstruktion von Typen. Die Code-Ausgabe ist zwar Solidity-ähnlich, allerdings ist der Output nicht direkt von einem Solidity-Compiler interpretierbar, worunter auch die Lesbarkeit leidet.

4.8 Hex-Rays

Hex-Rays [32] stellt abseits der Dekompilierung von EVM-Bytecode einen eher generischen Maschinencode-Decompiler dar. Dabei repräsentiert das Tool einen *De-facto-Standard* und übersetzt unter anderem die Maschinensprache *x86* in die höhere Programmiersprache *C*. Der Decompiler ist ähnlich der Literatur von Cifuentes [21] aufgebaut und führt somit nach der Erstellung von Zwischencode und CFG eine Daten- gefolgt von einer Kontrollflussanalyse durch. Für die Wiederherstellung von Kontrollstrukturen verwendet *Hex-Rays* eine strukturelle Analyse. Diese Methodik, welche mithilfe von *Pattern Matching* umgesetzt wird, hat den Nachteil, dass einige ursprüngliche Strukturen möglicherweise nicht erkannt werden können, womit eine sprungfreie Dekompilierung nicht garantiert werden kann. Anhand der Ergebnisse dieser Analysen wird der High-Level-Code zuerst als Pseudocode generiert, um danach durch mehrere Transformationen, Typanalysen und Umbenennungen an Lesbarkeit zu gewinnen und dadurch eine höhere Ähnlichkeit mit dem ursprünglichen Code der Zielsprache aufzuweisen.

4.9 Phoenix

Im Paper von Schwartz et al. [19] wird der Decompiler *Phoenix* vorgestellt, welcher laut den Verfassern 30-mal mehr Kontrollstrukturen wiederherstellen kann als Hex-Rays. Dabei wird bei der Dekompilierung anders als in vorausgehender Literatur ein höherer Wert auf die Korrektheit des Outputs sowie auf die verbesserte Wiederherstellung von Kontrollstrukturen gelegt. Ohne diese Eigenschaften kann eine Security-Analyse auf Basis der dekompilierten Ausgabe nicht verlässlich durchgeführt werden.

Ähnlich wie Hex-Rays setzt Phoenix dabei auf eine strukturelle Analyse, mit dem Unterschied, dass die genannten Themen speziell adressiert werden. Korrektheit wird mit dem Konzept *Semantics Preservation* garantiert. Dabei geht es darum, dass der Algorithmus, der bei der strukturellen Analyse angewandt wird, die ursprüngliche Semantik bewahrt „A structuring algorithm is semantics-preserving if it always transforms the input program to a functionally equivalent program representation“ – das dekompilierte Programm muss also semantisch äquivalent mit der ursprünglichen Applikation sein [19]. Die zweite wichtige Verbesserung, die Wiederherstellung von Kontrollstrukturen, wird mithilfe von *Iterative Refinement* umgesetzt. Entgegen aller früheren Decompiler beendet Phoenix die strukturelle Erkennung in einem bestimmten Bereich nicht, sobald eine Kontrollstruktur gefunden wurde. Stattdessen wird durch iterative Verfeinerung nach Hinweisen zu weiteren bzw. unterschiedlichen Strukturen gesucht. Dadurch wird

schlussendlich eine höhere Anzahl von Konstrukten, wie auch die `switch`-Anweisung erkannt, was zu weniger verbleibenden `goto`-Befehlen führt.

4.10 Dream

Im Gegensatz zu Hex-Rays [32] und Phoenix [19] setzt sich der Decompiler *Dream* [54] zum Ziel, von der strukturellen Analyse abzusehen, um beim dekompilierten Output ohne Sprünge auszukommen. Dies wird auch vom Paper mit dem Titel *No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations* verdeutlicht. Die verwendete Technik, die zu diesem Erfolg führt, ist im Gegensatz zum *Pattern Matching*, welches üblicherweise bei der strukturellen Analyse Anwendung findet, Pattern-unabhängig. Dies geschieht mithilfe des Konzepts *Pattern-Independent Structuring*, welches vermehrt auf der Semantik der High-Level-Strukturen und nicht auf der Form des zugrunde liegenden CFGs aufbaut. Der verwendete Algorithmus zur Strukturierung beruht auf der Annahme, das jedes High-Level-Konstrukt je genau einen identifizierbaren Einstiegs- sowie Ausstiegspunkt besitzt. Die sogenannten *Semantics-Preserving Transformations* sorgen dafür, dass diese Annahme zutrifft, indem eine Region mit mehreren Einstiegs- bzw. Ausstiegspunkten zu einem semantisch äquivalenten Konstrukt mit je genau einem transformiert wird. Dadurch können z. B. abnormale Schleifenaustritte wie die Anweisungen `break` und `continue` reproduziert werden. Außerdem werden nach Wiederherstellung des Kontrollflusses *Post-Structuring Optimizations* angewandt, mit welchen die Strukturen bzw. Lesbarkeit des resultierenden Codes verbessert werden können. So ist es beispielsweise möglich, eine `while`- zu einer `for`-Schleife zu transferieren, welche bei den ersten Dekompilierungsschritten üblicherweise nicht berücksichtigt wird, da jede Schleife mit `while` abgebildet werden kann.

KAPITEL 5

Ergebnisse

In diesem Kapitel werden die zentralen Ergebnisse der Diplomarbeit basierend auf dem methodischen Vorgehen präsentiert. Um den Aufbau von EVM-Bytecode im Detail zu verstehen, ist die Analyse solcher Low-Level-Codes unter der Betrachtung ihrer Solidity-Codes von Bedeutung. In einem nächsten Schritt werden alle relevanten und bekannten Forschungsarbeiten untersucht, welche sich bereits ansatzweise mit dem Thema *Decompilation of EVM Bytecode* beschäftigt haben. Hierfür werden die Prototypen, welche im Rahmen der Related Work aus Kapitel 4 entstanden sind, im Hinblick auf ihre Tauglichkeit zur Weiterverwendung in dieser Diplomarbeit detailliert analysiert und verglichen. Nicht zuletzt wird, aufbauend auf einem bereits bestehenden Tool, ein eigener Prototyp entwickelt, welcher versucht, EVM-Bytecode zurück in den ursprünglichen Code der höheren Programmiersprache Solidity zu übersetzen. Die dafür durchzuführenden Tätigkeiten werden ebenfalls in diesem Kapitel diskutiert. Eine Evaluierung des erstellten Prototyps mithilfe von realen, auf der Ethereum-Blockchain verfügbaren Daten darf hierbei nicht fehlen, um die Mängel bzw. potenziellen Verbesserungsmöglichkeiten des Prototyps zu identifizieren.

5.1 Aufbau von EVM-Bytecode

In den weiteren Abschnitten dieses Kapitels werden verschiedenste Decompiler diskutiert. Damit man die Tools anhand bestimmter Aspekte vergleichen kann, müssen diese auf einige reale Smart Contracts angewandt werden. Die folgenden Unterpunkte enthalten deshalb zwei Beispiele für Smart Contracts in Solidity, welche auf der Ethereum-Blockchain vorhanden sind. Danach wird der Aufbau des kompilierten EVM-Bytecodes näher analysiert, um die notwendigen Erkenntnisse zu gewinnen, die als Basis zur Beantwortung der Forschungsfragen dienen.

5.1.1 Ballot

Der Contract `Ballot.sol` stellt aufgrund seiner Komplexität und der Tatsache, dass der Quelltext zahlreiche Solidity-Features abdeckt, ein Paradebeispiel für einen Smart Contract dieser Sprache dar [12].

In diesem *Ballot* (siehe dazu Abbildung 5.1), welcher ein Online-Voting implementiert, geht es darum, den richtigen Personen ein Wahlrecht zu hinterlegen und dadurch Manipulationen zu vermeiden. Dabei ist es jedoch auch möglich, sein Wahlrecht zu delegieren – das Voting ist damit jedenfalls automatisch und transparent durchführbar.

Der Contract wird zuerst über den Konstruktor erzeugt. Dabei werden der Erzeuger als Vorsitzender des Votings und die spezifizierte Anzahl an Wahlvorschlägen gesetzt. In weiterer Folge kann der Wahlvorsitzende mittels eines Aufrufs der Funktion `giveRightToVote(address)` jedem Wähler, welcher noch nicht am Voting teilgenommen hat, ein Wahlrecht zuteilen. Der Wähler kann daraufhin sein Voting-Recht an eine andere Person delegieren (Funktion `delegate(address)`) oder seine Stimme zu einem existierenden Wahlvorschlag abgeben (Funktion `vote(uint8)`). Diese beiden Aufrufe sind jedoch nur möglich, wenn der Sender (noch) ein Voting-Recht besitzt. Bei beiden Funktionsaufrufen verfällt außerdem das Wahlrecht des Aufrufers. Die Funktion `winningProposal()` kalkuliert den stimmenstärksten Wahlvorschlag und kann somit am Ende des Votings zur Ermittlung des Wahlsiegers verwendet werden.

5.1.2 BasicToken

Im sechsteiligen Blog von Alejandro Santander namens *Decontructing a Solidity Contract* [46] wird der EVM-Bytecode eines in Solidity geschriebenen Smart Contracts in all seinen Einzelteilen analysiert. Dabei handelt es sich um den Contract `BasicToken.sol`, welcher aus einem Konstruktur und drei `public` Funktionen besteht. Der Source Code dazu findet sich in Abbildung 5.2.

Der Konstruktor initialisiert dabei das Gesamtangebot an Tokens, indem er die übergebene Anzahl in die globale Variable `totalSupply_` schreibt, sowie dem Erzeuger des Contracts, also dem `msg.sender`, die Gesamtanzahl an Tokens im Mapping `balances` zuordnet. Mithilfe der drei Funktionen ist es möglich, das Gesamtangebot an Tokens (Funktion `totalSupply()`) und die Anzahl der Tokens, die einem bestimmten Eigentümer zugeordnet sind, zu erhalten (Funktion `balanceOf(address)`) sowie einer beliebigen Adresse eine bestimmte Anzahl an Tokens zu übertragen (Funktion `transfer(address,uint256)`). In der zuletzt genannten Routine wird zudem sichergestellt, dass der Sender genug Tokens zur Verfügung hat.

Wird der Contract nun kompiliert, entsteht aus dem HLL-Code, geschrieben in der höheren Programmiersprache Solidity, der EVM-Bytecode. Hierbei handelt es sich um den Maschinencode, welcher aus hexadezimalen Werten besteht. Der kompilierte EVM-Bytecode des Contracts `BasicToken` findet sich in Abbildung 5.3.

```
contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        uint8 vote;
        address delegate;
    }
    struct Proposal {
        uint voteCount;
    }

    address chairperson;
    mapping(address => Voter) voters;
    Proposal[] proposals;

    constructor(uint8 _numProposals) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        proposals.length = _numProposals;
    }

    function giveRightToVote(address toVoter) public {
        if (msg.sender != chairperson || voters[toVoter].voted) return;
        voters[toVoter].weight = 1;
    }

    function delegate(address to) public {
        Voter storage sender = voters[msg.sender];
        if (sender.voted) return;
        while (voters[to].delegate != address(0)
                    && voters[to].delegate != msg.sender)
            to = voters[to].delegate;
        if (to == msg.sender) return;
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegateTo = voters[to];
        if (delegateTo.voted)
            proposals[delegateTo.vote].voteCount += sender.weight;
        else
            delegateTo.weight += sender.weight;
    }

    function vote(uint8 toProposal) public {
        Voter storage sender = voters[msg.sender];
        if (sender.voted || toProposal >= proposals.length) return;
        sender.voted = true;
        sender.vote = toProposal;
        proposals[toProposal].voteCount += sender.weight;
    }

    function winningProposal() public view returns (uint8 _winningProposal) {
        uint256 winningVoteCount = 0;
        for (uint8 prop = 0; prop < proposals.length; prop++)
            if (proposals[prop].voteCount > winningVoteCount) {
                winningVoteCount = proposals[prop].voteCount;
                _winningProposal = prop;
            }
    }
}
```

Abbildung 5.1: Solidity-Code des Smart Contracts *Ballot* [12]

```
contract BasicToken {

  uint256 totalSupply_;
  mapping(address => uint256) balances;

  constructor(uint256 _initialSupply) public {
    totalSupply_ = _initialSupply;
    balances[msg.sender] = _initialSupply;
  }

  function totalSupply() public view returns (uint256) {
    return totalSupply_;
  }

  function transfer(address _to, uint256 _value)
        public returns (bool) {

    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    return true;
  }

  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }
}
```

Abbildung 5.2: Solidity-Code des Smart Contracts *BasicToken* [46]

Um diesen Bytecode zu interpretieren, werden Operationscodes angewandt, zu welchen ein Mapping von Hex-Werten zu Mnemonics existiert. Jeder Opcode hat eine eindeutige ID im Wertebereich von `0x00` bis `0xFF`. Die Länge dieser Identifikation beträgt somit exakt ein Byte. Im Falle einer `PUSH`-Anweisung hat die Operation zusätzlich ein Argument der Länge von bis zu 32 Bytes. Alle anderen Instruktionen beziehen ihre Parameter vom Stack. Nach dem *Disassembling* anhand dieses Opcode-Mappings erhält man den Code in einer besser verständlicheren, textuellen Stacksprache (Auszug aus dem Disassembling siehe Abbildung 5.4).

Im folgenden Abschnitt wird der Aufbau des EVM-Bytecodes, im Speziellen des *Deployment-Codes*, näher beleuchtet.

5.1.3 Aufbau Deployment-Code

Der kompilierte EVM-Bytecode wird aufgrund der Tatsache, dass er bei der Erzeugung des Contracts ausgeführt wird, auch *Deployment-Code* genannt. Der Deployment-Code

```
6080604052348015610010576000 80fd5b506040516020806102178339810160409081
52905160008181553381526001602052919091205561 01d1806100466000396000f300
6080604052600436106100565763ffffffff7c010000000000000000000000000000000
0000000000000000000000000000600035041663 18160ddd811461005b57806370a08231
14610082578063a9059cbb146100b0575b600080fd5b348015610067576000 80fd5b50
6100706100f5565b604080519182525190819003602001 90f35b34801561008e576000
80fd5b5061007073ffffffffffffffffffffffffffffffffffffffff600435166100fb
565b3480156100bc57600080fd5b506100e173ffffffffffffffffffffffffffffffff
ffffffff60043516602435610123565b604080519115158252519081900360200190f3
5b60005490565b73ffffffffffffffffffffffffffffffffffffffff16600090815260
016020526040902054905490565b600073ffffffffffffffffffffffffffffffffffffffff
831615156101475760008 0fd5b336000908152600160205260409020548211156101 63
57600080fd5b503360009081526001602081905260408083208054859003905573ffff
ffffffffffffffffffffffffffffffffffff851683529091208054830190559291505 0
5600a165627a7a72305820a5d999f4459642872a29be93a490575d345e40fc91a7cccb
2cf29c88bcdaf3be0029
```

Abbildung 5.3: EVM-Bytecode *BasicToken*

```
000 PUSH1 80
002 PUSH1 40
004 MSTORE
005 CALLVALUE
006 DUP1
007 ISZERO
008 PUSH2 0010
011 JUMPI
012 PUSH1 00
014 DUP1
015 REVERT
016 JUMPDEST
017 POP
018 PUSH1 40
020 MLOAD
021 PUSH1 20
023 DUP1
024 PUSH2 0217
027 DUP4
028 CODECOPY
...
```

Abbildung 5.4: Auszug aus dem Disassembling des EVM-Bytecodes

eines Smart Contracts besteht aus *Creation-* und *Runtime-Code*, einem sogenannten *Metadata-Hash* und einem Abschnitt für Konstruktorargumente. Der Aufbau ist in Abbildung 5.5 visualisiert. Dabei ist jedenfalls zu beachten, dass diese Struktur nur für *einfache* Smart Contracts, welche keine weiteren Contracts über den Solidity-Befehl `new` bzw. die EVM-Operation `CREATE` erzeugen, gilt. Ist dies jedoch der Fall, kann der Deployment-Code aus beliebig vielen Iterationen oder Schachtelungen (abhängig von der verwendeten Solidity-Version) von Creation- bzw. Runtime-Programmen sowie Metadata-Hashes bestehen. Aus diesem Grund ist es notwendig, die einzelnen Teile des EVM-Bytecodes separat zu betrachten, anstatt den Deployment-Code im Gesamten. In den nächsten Abschnitten werden die einzelnen Bereiche daher detailliert erläutert.

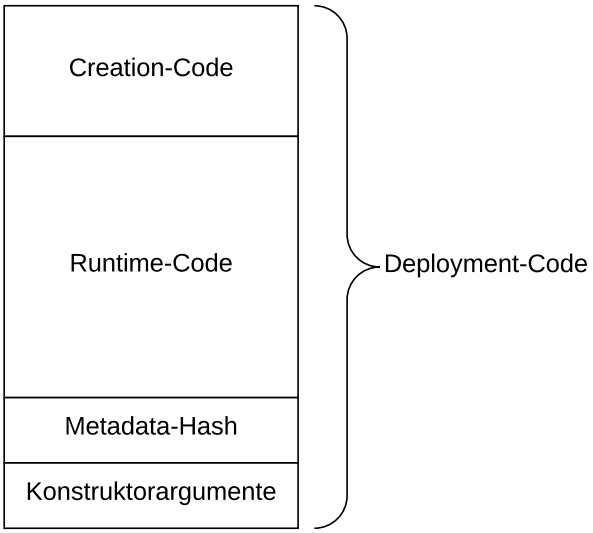

Abbildung 5.5: Aufbau eines EVM-Bytecodes

5.1.4 Creation-Code

Der Creation-Code entspricht dem Programm, welches bei der Erzeugung des Contracts ausgeführt wird. Dieser spezielle Teil des Bytecodes kann als Konstruktor des Smart Contracts gesehen werden. Als erste Aktion wird die Initialisierung des *Free Memory Pointers* durchgeführt, welche der Reservierung eines Memory-Abschnitts für die spätere Zwischenspeicherung von speziellen Elementen entspricht. Diese Tätigkeit wird am Beginn jedes Bytecode-Programms ausgeführt, unabhängig davon, ob es sich um das Creation- oder um das Runtime-Programm handelt. Nach der Durchführung von bestimmten Sicherheitsüberprüfungen werden gegebenenfalls Eingangsparameter aus dem bereits erwähnten Bereich des EVM-Bytecodes geholt, welcher die Konstruktorargumente enthält, und in das Memory geschrieben, damit die Parameter im Rumpf des Konstruktors verwendet werden können. Dieser nachfolgende Rumpf enthält den eigentlichen Code des ursprünglichen Solidity-Konstruktors. In diesem werden bspw. Variableninitialisierungen

durchgeführt, indem Werte mittels der EVM-Instruktion `SSTORE` in den Storage des Contracts geschrieben werden. Außerdem kann der Creation-Code dazu verwendet werden, den Contract nach der Erzeugung sofort wieder zu zerstören, was über den Befehl `SELFDESTRUCT` ermöglicht wird.

Vor dem Abschluss der Erzeugung des Contracts wird üblicherweise die Bytecode-Sequenz `CODECOPY PUSH1 00 RETURN` ausgeführt. Mithilfe dieser Befehlsfolge wird der Runtime-Code des soeben initialisierten Contracts zuerst in das Memory kopiert und daraufhin zur EVM retourniert. Diese speichert den Rückgabewert als den Code, der bei nachfolgenden Contract-Funktionsaufrufen ausgeführt werden soll. Dieser Schritt ist deshalb von Bedeutung, da aufgrund der separaten Betrachtung des Runtime-Codes dessen Programmadressen beim Wert `0` beginnen. Würden diese, wie im Deployment-Code, hinter dem Creation-Code fortlaufend nummeriert werden, könnten die Sprungziele im Runtime-Code nicht korrekt aufgelöst werden, da diese im Bytecode absolut zum Startwert `0` hartkodiert sind. Dies hätte zur Folge, dass die Sprünge im Bytecode-Programm nicht sinnvoll durchgeführt werden könnten, wodurch die Funktionen des Contracts nicht korrekt aufrufbar wären.

5.1.5 Runtime-Code

Nach der obligatorischen Initialisierung des *Free Memory Pointers* wird mithilfe des EVM-Opcodes `CALLDATASIZE` überprüft, ob eine bestimmte Funktion aufgerufen wurde. Falls diese Annahme zutrifft, wird zum *Function Selector* weitergegangen, um die aufgerufene Funktion zu bestimmen. Falls nicht wird zur *Fallback*-Routine gesprungen, welche aufgerufen wird, wenn keine passende Funktion existiert.

Der *Function Selector* beginnt mit dem Laden der Funktionssignatur, welche in der EVM durch den sogenannten *4-Byte-Hash* repräsentiert wird. Dieser Wert entspricht den ersten vier Bytes des Keccak-256-Hashes der Funktionssignatur. Nach diesem Prinzip korrespondiert der 4-Byte-Hash der Funktionssignatur `balanceOf(address)` mit dem hexadezimalen Wert `0x70a08231` [45]. Der *Function Selector* lädt den Hashwert mithilfe von `CALLDATALOAD` aus dem Calldata-Bereich. Der spezifizierte 4-Byte-Hash wird mit den hartkodierten Hash-Werten der öffentlichen Contract-Funktionen verglichen. Wird eine passende Funktion gefunden, wird zum jeweiligen Funktionsrumpf gesprungen, ansonsten verläuft die Ausführung wieder in die *Fallback*-Routine, welche keine Signatur besitzt und folglich als `function()` dargestellt wird.

```
keccak256("balanceOf(address)") =
70a08231b98ef4ca268c9cc3f6b4590e4bfec28280db06bb5d45e689f2a360be
```

Abbildung 5.6: Der Keccak-256-Hash der Funktionssignatur *balanceOf(address)*

Jede Funktion hat zusätzlich zu ihrem Rumpf, in welchem die eigentliche Funktionalität des ursprünglichen Codes ausgeführt wird, einen sogenannten *Function Wrapper*. Dieser ist vor der Ausführung des Rumpfes bspw. dafür zuständig, einen *Non-Payable-Check*

durchzuführen, falls die Funktion nicht mit dem Schlüsselwort `payable` markiert wurde. Diese Überprüfung bricht die Ausführung ab, falls zu einer Funktion, welche in Solidity nicht mit `payable` gekennzeichnet wurde, dennoch ETH übertragen werden. Außerdem werden, falls vorhanden, Parameterdaten aus dem Calldata-Bereich geholt und in die benötigte Form gebracht, damit diese während der Ausführung der eigentlichen Funktionalität verwendet werden können. Nach Ablauf dieser Tätigkeiten wird zum eigentlichen Funktionsrumpf gesprungen. Zum Abschluss der Ausführung des Rumpfes wird ein Sprung zum zweiten Teil des Wrappers durchgeführt, welcher dazu verwendet wird, die potenziellen Rückgabedaten der Funktion im entsprechenden Format zu retournieren.

Calldata

Wie bereits erwähnt hat der aufgerufene Contract die Möglichkeit, auf die Call-Daten zuzugreifen. Falls sich der Contract an den ABI-Standard hält, entsprechen die ersten vier Bytes des Calldata-Bereichs dem 4-Byte-Hash der aufgerufenen Funktion. Mithilfe dieser Signatur wird der auszuführende Codeteil identifiziert. In Abbildung 5.7 befinden sich die Call-Daten zu einem beispielhaften Aufruf der Funktion `balanceOf(address)` des Contracts `BasicToken`. Die ersten vier Bytes, also der hexadezimale Wert `0x70a08231`, stehen dabei für den 4-Byte-Hash der Funktionssignatur, die letzten 20 Bytes für den Parameter vom Typ `address`.

```
0x70a08231000000000000000000000000ca35b7d915458ef540ade6068dfe2f44e8fa733c
```

Abbildung 5.7: Calldata eines Funktionsaufrufes. Die ersten 4 Bytes stehen für den Hash der Funktionssignatur und die letzten 20 Bytes für den Adressparameter [46].

Unter Zuhilfenahme der Opcodes, welche in Tabelle 5.1 dargestellt sind, können die Daten aus diesem Abschnitt während der Ausführung verwendet werden.

Opcode	Mnemonic	Beschreibung
35	CALLDATALOAD	Legt einen Ausschnitt des Calldata-Bereichs in der Größe von 32 Bytes auf den Stack.
36	CALLDATASIZE	Legt die Größe des Calldata-Bereichs auf den Stack.
37	CALLDATACOPY	Kopiert einen bestimmten Ausschnitt des Calldata-Bereichs in das Memory.

Tabelle 5.1: Opcodes zur Verwendung der Transaktionsdaten

5.1.6 Metadata-Hash

Üblicherweise befindet sich nach dem Runtime-Code jedes Contracts ein Metadata-Hash, welcher, wie der Name schon erahnen lässt, den Hash der Metadaten des Contracts darstellt. Diese Daten sind im Gegensatz zum bisher analysierten Code nicht als Opcodes, sondern als rohe Bytes zu interpretieren. Die Interpretation dieser Metadaten als Opcodes

würde kein sinnvolles Programm ergeben und ferner kann dieser Bereich nicht über Sprünge aus dem Programmcode erreicht werden.

Der Metadata-Hash besitzt eine bestimmte Struktur und wird vom Solidity-Compiler in den Bytecode eingefügt. Mithilfe dieses Hashwertes kann in Swarm [14], einem dezentralen Speichersystem, nach den Metadaten des Contracts gesucht werden. Im Laufe der Solidity-Historie kamen für den Metadata-Hash verschiedene Formate zur Anwendung. Die beiden bekanntesten sind in den folgenden Abbildungen dargestellt. Die zweite Struktur (siehe Abbildung 5.9) unterscheidet sich demnach von der ersten (Abbildung 5.8), indem u. a. die Version des verwendeten Solidity-Compilers aufgenommen wurde.

```
0xa1 0x65 'b' 'z' 'z' 'r' '0' 0x58 0x20 <32 bytes swarm hash> 0x00 0x29
```

Abbildung 5.8: Format des Metadata-Hashes aus der Solidity-Dokumentation v0.5.8 [27]

```
0xa2
0x65 'b' 'z' 'z' 'r' '1' 0x58 0x20 <32 bytes swarm hash>
0x64 's' 'o' 'l' 'c' 0x43 <3 byte version encoding>
0x00 0x32
```

Abbildung 5.9: Format des Metadata-Hashes aus der Solidity-Dokumentation v0.5.13 [27]

5.1.7 Konstruktorargumente

Die Konstruktorargumente befinden sich üblicherweise am Ende des Deployment-Bytecodes und enthalten die Parameterdaten, die der Konstruktor für die Erzeugung des Contracts benötigt. Der Creation-Code kann mithilfe des Opcodes `CODECOPY` auf die entsprechende Sektion im Deployment-Code zugreifen. Wie der Metadata-Hash sind auch diese Daten nicht als Opcodes, sondern als rohe Bytes zu interpretieren. Wird für den Parameter `_initialSupply` des BasicToken-Contracts bspw. der Wert 10.000 übergeben, existiert ein Konstruktorargument mit dem hexadezimalen Wert `0x2710`. Da es sich beim betroffenen Parametertyp um `uint256` handelt, werden die vorangestellten 30 Bytes mit Nullen aufgefüllt.

```
0x0000000000000000000000000000000000000000000000000000000000002710
```

Abbildung 5.10: Konstruktorargument mit dem Wert 10.000 [46]

5.1.8 Diskussion

In den vorausgehenden Abschnitten wurde ersichtlich gemacht, worauf bei der Dekompilierung von EVM-Bytecode Rücksicht genommen werden muss. Der zu dekompilierende Bytecode, welcher unter Umständen einem Deployment-Code entspricht, kann nicht im Gesamten betrachtet, sondern muss in seine Einzelteile zerlegt werden.

Die Bereiche *Metadata-Hash* und *Konstruktorargumente* sind für die Dekompilierung überhaupt nicht relevant, da es sich in Bezug auf den Metadata-Hash um nicht benötigte Metadaten des Contracts und im Hinblick auf die Konstruktorargumente um Daten handelt, die keinen Mehrwert für die Rückübersetzung mit sich bringen. Somit wäre eine Heuristik sinnvoll, die diese Abschnitte aus dem EVM-Bytecode gänzlich entfernt, damit diese Daten bei der Dekompilierung nicht im Wege stehen.

Ein weiterer Aspekt ist das Auftreten der Creation- und Runtime-Programme im Deployment-Code. Da hier sowohl einfache Contracts, welche keine weiteren Contracts erzeugen, als auch solche, die weitere erstellen und dadurch mehrere Schachtelungen bzw. Wiederholungen der erwähnten Bytecodeabschnitte beinhalten, betroffen sind, ist eine Trennung in die einzelnen Teile von besonderer Bedeutung. Damit die Dekompilierung, insbesondere die Erstellung von Kontrollflussgraphen, korrekt durchgeführt werden kann, ist es zudem notwendig, jeden Codeteil separat zu betrachten. Dies geht mit der Tatsache einher, dass die Programmadressen der einzelnen Programmteile im Deployment-Code fortlaufend nummeriert werden. Um jedoch die hartkodierten Sprungziele und dadurch den Kontrollfluss wiederherstellen zu können, ist es essenziell, jedes ausführbare Programm separat, beginnend mit der Adresse `0` zu durchlaufen. Zusätzlich zur Entfernung der nicht benötigten Daten ist somit die Trennung in die einzelnen Teile notwendig, damit die korrekte Wiederherstellung des CFGs garantiert werden kann. Auch hier ist eine Heuristik unumgänglich, welche entweder den Beginn oder das Ende eines separat zu betrachtenden Bytecode-Programms identifiziert.

Die durch die Analyse der Struktur von EVM-Bytecode gewonnenen Erkenntnisse bilden die Grundlage zur Beantwortung der Forschungsfrage „Wie kann die Erstellung des CFGs für EVM-Bytecode-Programme verbessert werden?“. Diese wird in Abschnitt 5.3.1 detailliert behandelt.

5.2 Aufbau analysierter Forschungsarbeiten

In den folgenden Abschnitten werden die in Kapitel 4 erwähnten Tools zum Thema *Decompilation of EVM Bytecode* separat betrachtet und die jeweilige Eignung für die weiterführende Verwendung in dieser Diplomarbeit ermittelt. Die Entscheidung, auf welchem Tool letztendlich aufgebaut wird, wird anhand einer Gegenüberstellung begründet.

5.2.1 Oyente

Bei der Evaluierung des Programms *Oyente* [40] wurden einige Nachteile sichtbar, die das Tool für die Erzeugung von Solidity-Code auf Basis von EVM-Bytecode unzureichend machen. Zum einen wird die Visualisierung des CFGs, welche für die Beantwortung der ersten Forschungsfrage vorteilhaft wäre, nicht unterstützt. Zum anderen stammt das Paper bereits aus dem Jahr 2016, allerdings sind seit der Veröffentlichung des Tools noch zahlreiche Commits erfolgt, wodurch es keine genaue Beschreibung über den aktuellen Funktionsumfang von Oyente gibt. Nichtsdestotrotz liegt der letzte Commit bereits

mehr als ein Jahr zurück, weswegen neuere Solidity-Versionen nicht mehr verwendbar sind. Die letzte unterstützte Version ist 0.4.19, die aktuelle 0.5.13 (Stand: Dezember 2019). Außerdem wurde das Programm, wie bereits erwähnt, für die Erkennung von Sicherheitslücken in Smart Contracts, anstatt für Reverse Engineering entwickelt. Diese Sicherheitsüberprüfung wird üblicherweise auf den Solidity-Code des Contracts angewandt, wodurch die Möglichkeit zur Eingabe von EVM-Bytecode nur in der Theorie besteht, in der Praxis allerdings zu Fehlern führt. Die größte Einschränkung von Oyente für diese Arbeit ist deshalb die Tatsache, dass das Programm nur mit dem Input von Solidity-Code, nicht aber mit EVM-Bytecode sinnvolle Ergebnisse liefert.

5.2.2 Mythril

In *Mythril* [25] wird der Kontrollflussgraph im Gegensatz zu Oyente auch visualisiert. Der Graph wird dabei als HTML dargestellt, wobei die Knoten Blöcke, welche die einzelnen Befehle beinhalten und die Kanten Sprünge zwischen den Blöcken darstellen. Zusätzlich werden auf den Kanten die Bedingungen angezeigt, die eintreffen müssen, damit ein bedingter Sprung durchgeführt wird. Diese detaillierte Darstellung hat allerdings zur Folge, dass der CFG unübersichtlich und damit für die Analyse von komplexeren Contracts ungeeignet ist. Ein weiterer Nachteil des Tools in Bezug auf diese Diplomarbeit ist die Tatsache, dass das Anwendungsgebiet die Untersuchung von Contracts auf Sicherheitslücken ist und Myhtril deshalb keinen Code, weder als Solidity- noch als Zwischenrepräsentation, erzeugt. Zudem ist die Unterscheidung zwischen Creation- und Runtime-Code zwar theoretisch möglich, praktisch kann das Programm allerdings nicht sinnvoll mit einem Bytecode, bestehend aus mehreren, separat auszuführenden Teilen umgehen.

Zusammengefasst ist die Entscheidung gegen Mythril also nicht nur deshalb gefallen, weil das Programm kein Reverse Engineering betreibt und sich stattdessen auf die Erkennung von Security-Bugs spezialisiert, sondern auch, weil die Verwendung von Mythril im Zuge der Generierung von Solidity-Code aus EVM-Bytecode mit unnötigem Aufwand in Verbindung gestanden wäre. So ist es nicht nur die unübersichtliche Visualisierung des CFGs, welche die für diese Arbeit notwendige manuelle Analyse von Smart Contracts erschwert, sondern auch die Tatsache, dass Mythril mit ca. 9.000 LOC zu den größten der analysierten Tools zählt. Ein Programm, welches dieselbe oder gar weiter fortgeschrittene Arbeit hinsichtlich Dekompilierung von EVM-Bytecode mit weniger Aufwand erledigt, ist somit für diese Diplomarbeit von höherer Relevanz.

5.2.3 Porosity

Tatsächlich ist es den Entwicklern des Tools *Porosity* [24] gelungen, einen Decompiler zu implementieren, welcher EVM-Bytecode zurück in Solidity-ähnlichen Code übersetzt. Die essenzielle Einschränkung besteht allerdings darin, dass die Dekompilierung nur für einen sehr geringen Teil aller Contracts funktioniert und zwar nur für solche mit wenig Funktionalität. Die Code-Reproduktion ist deshalb auf bestimmte Solidity-

Strukturen beschränkt. Wie das Ergebnis der Tool-Analyse verdeutlicht, werden zum Beispiel Funktionen und auch Storage-Zugriffe erkannt. Rückgabewerte sind teilweise auch noch erkennbar, bei bedingten Anweisungen funktioniert die Detektion nicht mehr einwandfrei. Überhaupt nicht dekompiliert werden beispielsweise Konstruktoren, Schleifen, Events und `require`-Instruktionen. Außerdem gibt das Programm für Contracts aktueller Solidity-Versionen eine Fehlermeldung aus, welche besagt, dass bestimmte Opcodes nicht dekompiliert werden können, da diese nicht implementiert wurden. Dieses Problem geht wohl unter anderem mit der Tatsache einher, dass gewisse Opcodes, wie zum Beispiel `REVERT`, `STATICCALL`, `RETURNDATASIZE` und `RETURNDATACOPY` erst in späteren EVM-Versionen hinzugefügt wurden und die Entwicklung von Porosity schon im Jahre 2017 eingestellt wurde. Aber auch früher existierende Befehle wie `BALANCE` oder `CALLDATASIZE` werden nicht unterstützt. Eine weitere Einschränkung ist, dass Porosity für die Erkennung von Funktionssignaturen keine externe Bibliothek oder Datenbank verwendet, sondern das Erkennen von Funktionsnamen nur dann ermöglicht, wenn das Application Binary Interface (ABI) des Contracts als Parameter zur Ausführung angegeben wird. Dieses ABI steht allerdings, genauso wie der Solidity-Code des Contracts, nur selten zur Verfügung, weshalb die Dekompilierung von EVM-Bytecode überhaupt erst an Relevanz gewinnt. Das Fehlen dieser wichtigen Sprachelemente rückt Porosity für die weitere Verwendung in dieser Diplomarbeit in ein schlechtes Licht.

5.2.4 Erays

Erays [51] ist im Hinblick auf Reverse Engineering von Ethereum Smart Contracts das am weitesten fortgeschrittene Tool. Im Vergleich zu Porosity, welches lediglich ca. 5 % der auf der Ethereum Blockchain verfügbaren Contracts dekompilieren kann, übersetzt Erays mehr als 97 % [55] und das noch dazu in angemessener Zeit.

Die Nachteile des Tools sind, dass der produzierte High-Level-Code dem von Solidity wenig bis gar nicht entspricht. Verbesserungspotenzial gibt es außerdem bei der Erkennung von Strukturen. Bedingte Anweisungen und Schleifen werden erkannt, allerdings nur bis zu einem gewissen Grad. Bei den nicht-erkennbaren Strukturen verbleiben mehrere Sprünge und somit mehrere Blöcke in der PDF-Darstellung der jeweiligen Funktion. Letzteres stellt einen weiteren Nachteil dar: Funktionen werden zwar erkannt und unterschieden, diese werden allerdings unübersichtlich in eigene PDF-Dateien gepackt, welche noch dazu mit einem schwer zu identifizierenden Namen versehen sind. Eher kleinere Mängel sind die Nicht-Unterscheidung zwischen `require` und `assert` sowie zwischen Creation- und Runtime-Code. Außerdem besitzt das Tool keine Erkennung von Typen bzw. Events. Private Funktionen werden dagegen mithilfe einer Heuristik angenähert.

5.2.5 Vandal

Vandal [48] führt ein Disassembling des EVM-Bytecodes durch und baut einen CFG auf. Die Sprünge werden teilweise statisch ermittelt, teilweise dynamisch durch mehrere Iterationen über den gesamten CFG, wodurch die Graphen großer Contracts nicht

oder zumindest nicht mehr vollständig aufgebaut werden können. An dieser Stelle hat Vandal, im Gegensatz zu anderen untersuchten Programmen, mit massiven Performance-Problemen zu kämpfen. Für kleine bis mittelgroße Contracts wird der CFG jedoch korrekt und halbwegs effizient erzeugt, sofern es sich beim spezifizierten EVM-Bytecode lediglich um den Runtime-Code handelt und der Contract keine neuen Contracts erzeugt. Wird das Tool allerdings auf den Deployment-Code angewandt, stimmen die Sprungziele in den jeweiligen Bytecode-Programmen nicht mehr überein, da Vandal die separaten Codeteile nicht als solche erkennt und etwaige neue Contracts nicht ab der Programmadresse 0 abarbeitet, sondern ihre ursprüngliche absolute Position im Deployment-Code verwendet.

Vorteile von Vandal sind die Features, mit denen die Funktionserkennung sowie die Visualisierung des Kontrollflussgraphen durchgeführt werden. Mithilfe der CFG-Visualierung ist die manuelle Analyse des EVM-Bytecodes möglich, da mittels einer Markierung die Untersuchung von einzelnen Funktionen bewerkstelligt werden kann. Außerdem wird der Code zusätzlich zur stackbasierten EVM-Sprache auch in einer registerbasierten Darstellung repräsentiert. Des Weiteren wird wie auch in Erays versucht, private Funktionen mithilfe einer eigenen Heuristik anzunähern. Die angesprochenen Funktionalitäten sind allerdings mit Vorsicht zu genießen, da für große Contracts mit etlichen Verzweigungen die Funktionsmarkierung nicht einwandfrei durchgeführt wird. An dieser Stelle macht sich die mangelhafte Reproduktion der indirekten Sprünge bemerkbar, wodurch nicht alle Contract-Funktionen korrekt abgebildet werden können.

5.2.6 teEther

Die Analyse ergab, dass das Programm *teEther* [23] prädestiniert dafür ist, einen relativ vollständigen CFG eines Contracts anhand seines Bytecodes zu berechnen. Was für kleine bis mittelgroße Contracts noch effizient durchgeführt werden kann, ist allerdings für große nicht mehr möglich. Dies wird auch vom Paper [37] unterstrichen, in welchem erwähnt wird, dass für 85 % der Contracts, die sich auf der Ethereum-Blockchain befinden, die Analyse innerhalb des vom Tool vorgegebenen 30-Minuten-Limits abgeschlossen werden konnte. Die Dekompilierung könnte also nicht für alle Contracts durchgeführt werden. Dies geht wohl mit der Tatsache einher, dass der CFG mithilfe eines Path-Finding-Algorithmus aufgebaut wird, um die Kanten, abhängig von den Vorgänger-Blöcken, richtig zu erkennen und unmögliche Pfade auszuschließen.

Der Code von teEther ist nicht unnötig aufgebläht und relativ leicht verständlich, was auch die fehlende Dokumentation aufwiegt. Der Nachteil des relativ knappen Codes ist jedoch, dass einige wesentliche, für die Dekompilierung benötige Features fehlen. So gibt es im Tool beispielsweise keine Funktionserkennung, keine Metadata-Hash-Detektion und auch keine Unterscheidung zwischen Creation- und Runtime-Bytecode. Der weit größere Nachteil ist jedoch, dass die dynamischen Stack-Abbilder nicht pro Instruktion gespeichert werden, so wie es etwa Erays macht, was ein Vorteil für die Code-Reproduktion wäre.

5.2.7 DSol

Auch das von Thomas Hybel entwickelt Programm *DSol* [33] schafft es, EVM-Bytecode in Solidity-ähnlichen Code zu dekompilieren. Dabei werden Konstruktor, Funktionen und Metadata-Hashes sowie auch bedingte Anweisungen und Schleifen grundsätzlich erkannt.

Allerdings besitzt das Tool auch einige Mängel, wodurch kein kompilierbarer HLL-Code wiederhergestellt werden kann. DSol erkennt zwar grundsätzlich die Funktionen des Contracts, stellt allerdings den vom Solidity-Compiler eingefügten Funktionsselektor auch im dekompilierten Code dar. Auch die Code-Unterscheidung ist nur für Contracts möglich, die maximal einen Creation- bzw. Runtime-Code beinhalten. Zudem wird vom Tool keine Erkennung der Funktions- bzw. Eventsignaturen durchgeführt. Ferner fehlt auch hier die Typisierung gänzlich. Die Unterscheidung zwischen `require` und `assert` sowie die sprungfreie Reproduktion von Kontrollstrukturen ist wie in Erays auch hier nicht gegeben. Eine essenzielle Einschränkung im Hinblick auf die Performance der Dekompilierung ist außerdem die Tatsache, dass nicht alle Test-Contracts fehlerfrei dekompiliert werden konnten.

5.2.8 Online-Decompiler

Des Weiteren wurden im Rahmen der Recherche zwei Online-Decompiler gefunden, von welchen weder ein Schriftstück existeirt, noch der Source Code öffentlich verfügbar ist. Diese Tools wurden auf den Bytecode von bekannten Contracts angewandt und hinsichtlich ihres Outputs analysiert.

ethervm.io

Der *Online Solidity Compiler* der Platform *ethervm.io* [31] kann wie die meisten Tools nicht mit der Kombination von Creation- und Runtime-Bytecode umgehen. Dekompiliert wird deshalb immer nur der erste Teil. Wendet man den Decompiler auf den Ballot-Contract an, erkennt man sofort, dass einige Sprünge nicht aufgelöst werden können. So werden zum Beispiel teilweise bedingte Anweisungen wiederhergestellt, nicht jedoch Schleifen. Es verbleiben zahlreiche `goto`-Instruktionen im rückübersetzten Code, welche auf definierte Labels springen. Des Weiteren existiert eine Funktion `main`, welche den Funktionsselektor repräsentiert. In diesem wird mithilfe des 4-Byte-Hashes geprüft, welche Funktion aufgerufen wurde und daraufhin die jeweilige Methode mit den benötigten Parametern ausgeführt. Dieser Funktionsselektor wird bekanntlich vom Solidity-Compiler erstellt, damit die EVM entscheiden kann, welche Contract-Funktion ausgeführt werden soll und ist somit nicht Teil des ursprünglichen Solidity-Codes. Positiv hervorzuheben ist, dass die Funktionssignaturen teilweise wiederhergestellt werden können. Diese werden anhand des 4-Byte-Hashes von der Online-Datenbank *4byte.directory* [4] bezogen, welche am Tag des Aufrufs knapp 150.000 Signaturen beinhaltete. Verbesserungspotenzial gibt es ferner bei der Darstellung von Variablen, sowie bei Storage- und Memory-Zugriffen. Diese werden relativ unstrukturiert und mit zahlreichen überflüssigen Operationen wiederhergestellt. `require`-Anweisungen werden vom Decompiler überhaupt nicht erkannt, sondern

als `if`-Ausdrücke mit anschließendem `revert`-Befehl dargestellt. Außerdem fehlt die Typisierung vollständig. Abbildung 5.11 zeigt die angeführten Nachteile in Form des dekompilierten Outputs der Funktion `giveRightToVote(address)` des Contracts `Ballot`.

```
function giveRightToVote(var arg0) {
    var var0 = msg.sender != storage[0x00] & 0x02 ** 0xa0 - 0x01;

    if (!var0) {
        memory[0x00:0x20] = arg0 & 0x02 ** 0xa0 - 0x01;
        memory[0x20:0x40] = 0x01;

        if (!(storage[keccak256(memory[0x00:0x40]) + 0x01] & 0xff)) {
            goto label_02FB;
        } else { goto label_02F7; }
    } else if (!var0) {
    label_02FB:
        memory[0x00:0x20] = arg0 & 0x02 ** 0xa0 - 0x01;
        memory[0x20:0x40] = 0x01;
        storage[keccak256(memory[0x00:0x40])] = 0x01;
        return;
    } else {
    label_02F7:
        return;
    }
}
```

Abbildung 5.11: Auszug aus dem Output des *Online Solidity Decompilers* von *ethervm.io* angewandt auf den Smart Contract *Ballot*

etherscan.io

Auch das Online-Portal *etherscan.io* besitzt ein Reverse-Engineering-Tool, den sogenannten *EVM bytecode decompiler* [28]. Die Analyse bezieht sich auf den Decompiler, welcher bis Mitte November 2019 verwendet wurde. Danach wurde der Decompiler ausgetauscht.

Ähnlich dem Programm von ethervm.io werden die externen Funktionen des Contracts erkannt und gewisse Funktionssignaturen wiederhergestellt. Auch die Identifizierung von bedingten Anweisungen ist möglich, wobei das Tool von Etherscan einen anderen Ansatz zu verfolgen scheint. Fakt ist nämlich, dass offensichtlich jeder mögliche Pfad des CFGs durchlaufen wird, also dass Teilbäume mehrfach traversiert werden, was dazu führt, dass einerseits zahlreiche geschachtelte `if-else`-Ausdrücke im reproduzierten Code auftauchen und andererseits etliche unnötige Codeduplikate vorkommen. Ansatzweise ist dieses Problem in Abbildung 5.12 erkennbar, welches die dekompilierte Funktion `vote(uint8)` des Smart Contracts `Ballot` zeigt. Diese naive Technik wurde auch in der aktuellen Diplomarbeit getestet, aber aufgrund ihrer Ineffizienz verworfen. Schleifen

werden wie auch bei ethervm.io nicht erkannt und aus diesem Grund verbleiben zahlreiche `goto`-Befehle im dekompilierten Code.

```
function vote(uint8 _arg0) public {
    ifmapping1[msg.sender][1] {
        if(mapping1[msg.sender][1] == 0) {
            mapping1[msg.sender] = ((_arg0 * 100) ||
                (-ff01 && (1 || (-100 && mapping1[msg.sender][1]))));
            require((_arg0 < storage[2]));
            storage[(keccak256(2) + _arg0)] += mapping1[msg.sender];
            return;
        } else {
            return;
        }
    } else {
        if(_arg0 < storage[2]) {
            mapping1[msg.sender] = ((_arg0 * 100) ||
                (-ff01 && (1 || (-100 && mapping1[msg.sender][1]))));
            require((_arg0 < storage[2]));
            storage[(keccak256(2) + _arg0)] += mapping1[msg.sender];
            return;
        } else {
            return;
        }
    }
}
```

Abbildung 5.12: Auszug aus dem Output des *EVM bytecode decompiler* von *etherscan.io* angewandt auf den Smart Contract *Ballot*

Die Vorteile von Etherscan sind hingegen, dass Variablen und Storage-Zugriffe relativ übersichtlich wiederhergestellt werden können. Bei Token-Contracts werden zusätzlich zu Funktions- auch Eventsignaturen erkannt. Erstaunlicherweise ist bei diesen Contracts außerdem die Wiederherstellung von Namen und teilweise auch Typen von Storage-Variablen möglich – vermutlich da sich Contracts dieser Art sehr ähneln. Der Decompiler reproduziert außerdem `require`-Anweisungen und erkennt ob eine Funktion mit dem Schlüsselwort `payable` versehen ist. Dagegen werden jedoch `assert`-Befehle nicht detektiert und stattdessen mit der Fehler-Anweisung `revert("Invalid instruction (0xfe)");` ersetzt.

5.2.9 Diskussion

In Tabelle 5.2 werden die analysierten Tools anhand relevanter Kriterien vergleichen. Dabei handelt es sich um folgende Metriken:

- **Disassembling**: Das Tool führt ein Disassembling des EVM-Bytecodes durch.

- **CFG-Erstellung**: Das Tool erstellt einen CFG auf Basis des EVM-Bytecodes.
- **EVM-Bytecode-Eingabe**: Das Tool ermöglicht eine fehlerfreie Abarbeitung des eingegebenen EVM-Bytecodes.
- **High-Level-Code-Ausgabe**: Das Tool erstellt Code einer höheren (Pseudo-) Programmiersprache.
- **Funktionserkennung**: Das Tool führt eine Funktionserkennung durch.
- **Metadata-Hash-Erkennung**: Das Tool erkennt Metadata-Hashes im Bytecode und entfernt diese.
- **Code-Unterscheidung**: Das Tool unterscheidet zwischen Creation- und Runtime-Code und kann mit beiden Arten gleichzeitig umgehen.
- **Open Source**: Der Source Code des Tools ist öffentlich zugänglich und aufgrund der Lizenzierung wiederverwendbar.

Tool	Disassembling	CFG-Erstellung	EVM-Bytecode-Eingabe	High-Level-Code-Ausgabe	Funktionenserkennung	Metadata-Hash-Erkennung	Code-Unterscheidung	Open Source
DSol	✓	✓	✓	✓	✓	✓	✓	✓
Erays	✓	✓	✓	✓	✓	✓		✓
etherscan.io	✓	✓	✓	✓	✓	✓	✓	
ethervm.io	✓	✓	✓	✓	✓	✓		
Mythril	✓	✓	✓		✓	✓		✓
Oyente	✓	✓			✓			✓
Porosity	✓	✓	✓	✓	✓			✓
teEther	✓	✓	✓					✓
Vandal	✓	✓	✓		✓			✓

Tabelle 5.2: Gegenüberstellung der analysierten Decompiling-Tools

Dabei führen alle analysierten Tools ein Disassembling und eine CFG-Erstellung durch, was Voraussetzung für die Dekompilierung ist. Auch die Eingabe von EVM-Bytecode wird selbstverständlich benötigt. Damit kann zwar theoretisch jedes untersuchte Programm umgehen, praktisch ist die fehlerfreie Abarbeitung bei *Oyente* jedoch nicht möglich, womit dieses quasi ausscheidet.

Die Features zur Ausgabe von High-Level-Code, Funktionserkennung, Metadata-Hash-Erkennung sowie Unterscheidung von Creation- und Runtime-Code sind zwar keine Ausschlusskriterien, von Vorteil sind diese Attribute allerdings schon. Aus diesem Grund wurde ein besonderer Blick auf *DSol*, *Erays*, *Porosity* sowie die beiden Online-Decompiler geworfen. Die Tools *Mythril* (zahlreiche überflüssige bzw. fehlende essenzielle Funktionalitäten), *teEther* (absente wesentliche Features) sowie *Vandal* (schlechte Performance bzw. fehlende Funktionalitäten) wären aufgrund der jeweiligen Probleme nur mit erhöhtem Aufwand für die weitere Verwendung in dieser Diplomarbeit verbunden und werden deshalb nicht weiter betrachtet.

Ferner sind jene Programme von höherer Relevanz, die im Rahmen dieser Diplomarbeit verwendet werden könnten, also von welchen der Source Code öffentlich zugänglich und aufgrund der Lizenzierung wiederverwendbar ist. Die beiden Online-Decompiler der Platformen *etherscan.io* bzw. *ethervm.io* sind aus dem Grund, dass ihr Code nicht als *Open Source* verfügbar ist, nicht zur weiteren Verwendung tauglich.

Trotz der Tatsache, dass die übrigen Tools im Vergleich ähnliche Features aufweisen, sind DSol und Erays wesentlich ausgereifter. Diese Annahme wird durch die Tatsache untermauert, dass bei Erays im Vergleich zu Porosity 97 % statt 5 % aller eindeutigen Contracts, die sich auf der Ethereum-Blockchain befinden, dekompilierbar sind [55]. Da auch die Analyse von Porosity wesentliche Mängel in der Dekompilierung aufzeigte und der Entwickler mehr oder weniger aufgab [24], ist das Tool für die weitere Verwendung ungeeignet.

Die Entscheidung zwischen Erays und DSol stellte sich als schwierig heraus, da beide Tools relativ fortgeschritten sind. DSol trifft zwar in der Gegenüberstellung alle Metriken und produziert bereits Solidity-ähnlichen Code, hat aber im Vergleich zu Erays wie erwähnt mit Performance-Einbußen zu kämpfen. Vor allem bei der Dekompilierung von großen Contracts, bei welcher Erays in nur wenigen Sekunden ein brauchbares Ergebnis liefert, liegt die Dauer bei DSol im Minutenbereich. Außerdem ist ein weiteres Mal zu betonen, dass Erays laut Paper ca. 97 % der Blockchain-Contracts dekompilieren kann, wohingegen DSol mit einigen Test-Contracts überhaupt nicht umgehen konnte.

Auf Grundlage dieser Überlegungen wurde entschieden, dass die wenigen Vorteile von DSol gegenüber Erays hinsichtlich Darstellung nicht die vergleichsweise hohen Performance-Einschränkungen aufwiegen können, womit die Entscheidung für Erays und gegen DSol gefallen ist.

5.3 Prototyp zur Dekompilierung

Im folgenden Abschnitt wird der im Rahmen dieser Diplomarbeit entwickelte Prototyp zum Thema *Decompilation of EVM Bytecode* im Detail beschrieben. Es wird außerdem eine Verbindung zu den Forschungsfragen gezogen und erläutert, welche Rolle dieser Prototyp bei der Bearbeitung der Fragen eingenommen hat.

Die Architektur des Prototyps ist in Abbildung 5.13 ersichtlich. Für die Behandlung der Forschungsfrage, wie die CFG-Erstellung auf Basis von EVM-Bytecode verbessert werden kann, wird ein Tool implementiert, welches den Bytecode in separat auszuführende Contract-Teile zerlegt. Näheres zum *Bytecode Splitter* findet sich in Abschnitt 5.3.1.

Diese isolierten Contract-Bytecodes, die vom *Bytecode Splitter* retourniert werden, stellen den Input des zweiten Forschungsparts dieser Arbeit dar. Für die Behandlung der Fragen, mit welchen Methoden die Dekompilierung von EVM-Bytecode zu Solidity-Code durchgeführt und wie viel vom ursprünglichen High-Level-Code mit den angewandten Techniken wiederhergestellt werden kann, wurde auf Grundlage des Reverse-Engineering-Tools *Erays* ein Prototyp mit dem Titel *Soldec*, kurz für *Solidity Decompiler*, erstellt. Der gesamte Prototyp wurde in der dritten Version der dynamischen Programmiersprache *Python* [10] implementiert.

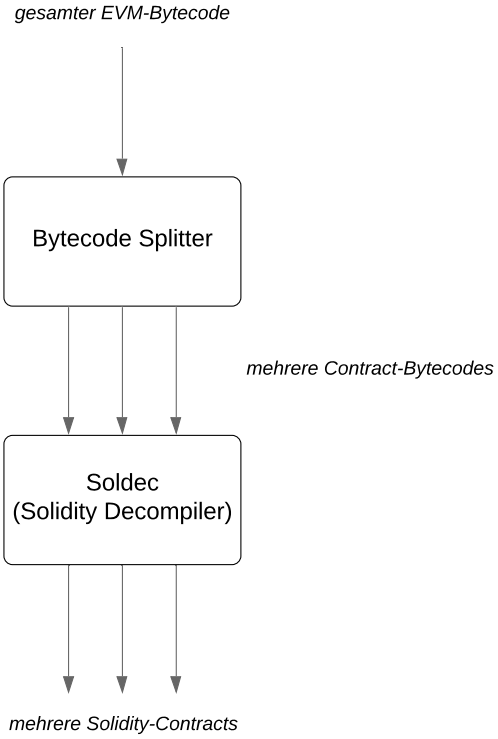

Abbildung 5.13: Architekur des Prototyps zur Dekompilierung von EVM-Bytecode

5.3.1 RQ1: Methoden zur CFG-Optimierung

Dieser Abschnitt behandelt die Forschungsfrage „Wie kann die Erstellung des CFGs für EVM-Bytecode-Programme verbessert werden?“. Tatsache ist, dass der Großteil der analysierten Open-Source-Programme, unabhängig davon, ob es sich um ein Reverse-Engineering-Tool oder ein Sicherheitsanalyseprogramm handelt, nicht mit dem Deployment-Code, also der gleichzeitigen Untersuchung von Creation- und Runtime-Code, umgehen

kann. Bei den meisten Tools gibt es dafür außerdem keine Auswahlmöglichkeit, es wird immer der erste und somit im Falle des Deployment-Codes immer der Creation-Code zur Abarbeitung verwendet.

Die auftauchenden Folgen lassen sich einfach darstellen. Will man ein solches Programm auf den Runtime-Code eines Contracts anwenden, ist man gezwungen, den Creation-Code manuell zu entfernen. Dieses Vorhaben ist natürlich möglich, jedoch erstens mit zusätzlichem Aufwand verbunden und zweitens fehleranfällig, da der EVM-Bytecode bekanntlich nur schwer für den Menschen lesbar ist und man daher die Bedeutung der Opcodes ohne Zusammenhang nicht immer hundertprozentig durchschauen kann.

Führt man die manuelle Trennung der nicht benötigten Daten allerdings nicht durch, wird man in manchen Programmen, die eine CFG-Erstellung ermöglichen, mit einem fehlerhaften Graphen konfrontiert. Dies ist darauf zurückzuführen, dass im Deployment-Code die Programmadressen der Runtime-Codes, wie bereits erwähnt, nicht mit den tatsächlichen Ausführungsadressen übereinstimmen. Dadurch erhält man einen relativ sinnfreien Graphen, in welchem die Verbindungen zwischen den einzelnen Basic Blocks nicht korrekt dargestellt werden, wodurch der CFGs unbrauchbar wird.

Das erwähnte Problem tritt des Weiteren bei Contracts ein, die neue Contracts über den Befehl `new` erzeugen. Auch der Code der neuen Contracts ist üblicherweise im Deployment-Code inkludiert, wodurch diese aufgrund der Nichtberücksichtigung des gesamten Codeteils oder des Offsets bei den Sprungadressen nicht sinnvoll reproduziert werden können.

Bytecode Splitter

Um diese Forschungsfrage zu bearbeiten, wird aus den genannten Gründen ein Prototyp umgesetzt, welcher gegebenenfalls die angesprochene Trennung des EVM-Bytecodes in separat auszuführende Contract-Teile durchführt und diese in weiterer Folge als einzelne Bytecode-Programme in das Dekompilierungsmodul weiterreicht.

Der sogenannte *Bytecode Splitter* verwendet EVM-Bytecode als Input. Hierbei ist es nicht von Bedeutung, ob es sich um den Deployment-, Creation- oder Runtime-Code eines Contracts handelt. Das Programm beginnt mit der Durchsuchung des spezifizierten Bytecodes auf Vorkommnisse des Metadata-Hashes. Werden solche Passagen gefunden, entfernt das Tool die betroffenen Abschnitte sofort aus dem Bytecode. Danach wird auf Basis des EVM-Opcode-Mappings ein Disassembling durchgeführt, indem hexadezimale Werte zu Mnemonics und ihren Parametern übersetzt werden. Währenddessen wird in einer Schleife anhand einer bestimmten Heuristik nach einem neuen, separat auszuführenden Programmstück gesucht. Falls der Algorithmus ein solches detektiert, wird der Bytecode aufgeteilt und die Programmadressen des neuen Teils mit dem Wert `0` beginnend zurückgesetzt. Der Quellcode der Schleife wird so lange zur Suche von neuen Programmteilen auf den verbleibenden Bytecode angewandt, bis das Ende erreicht ist, also der Bytecode keine weiteren Zeichen beinhaltet. Falls die Heuristik während der gesamten Ausführung kein neues Codestück erkennen konnte, ist der EVM-Bytecode des

Eingangsparameters bereits atomar ausführbar. Zum Abschluss ermittelt der Splitter, bei welchem der analysierten Bytecode-Programme es sich um einen Creation-Code handelt. Auch diese Determination wird anhand einer bestimmten Befehlsfolge gelöst.

Zur Trennung des Bytecodes existieren mehrere mögliche Heuristiken. Die im Rahmen dieser Arbeit untersuchten Verfahren werden in den nächsten Abschnitten genauer erläutert. Ferner werden die verwendeten Methodiken zur Entfernung des Metadata-Hashes bzw. zur Bestimmung der Creation-Codes beschrieben.

Heuristik zur Trennung des Bytecodes: Programmende bestimmen

Eine mögliche Heuristik zum Aufteilen des Bytecodes in einzelne Contract-Teile ist das Bestimmen des Endes eines separat auszuführenden Programmstücks. Der Ansatz teilt also den EVM-Bytecode in einzelne Teile, sofern das Ende eines Programms erreicht ist. Dieses Ende wird approximiert, indem auf die Opcodes `STOP` und `INVALID`, welche eine EVM-Ausführung terminieren, geachtet wird. Steht unmittelbar nach einem der beiden genannten Operationscodes kein `JUMPDEST`-Befehl, also ist die darauffolgende Anweisung kein Sprungziel, ist davon auszugehen, dass des Programm zu Ende ist und danach ein neuer Part beginnt. Diese Annahme beruht auf der Tatsache, dass Befehle, die nach einer Programmterminierung auftreten, nicht ausführbar sind, sofern sie nicht über Sprünge erreichbar sind. Da letztere mit dem Opcode `JUMPDEST` gekennzeichnet werden und in dieser Arbeit davon ausgegangen wird, dass *Dead Code* nur in Form von Call- sowie Metadaten, nicht aber während eines ausführbaren Codestücks auftreten kann, wird diese Heuristik als relevant angesehen.

Abbildung 5.14 zeigt einen Auszug aus dem Output des *Bytecode Splitters* unter Anwendung dieser Methodik. Der `STOP`-Befehl an der Adresse 191 wird vom Tool als Abschluss eines Programms erkannt, da unmittelbar danach keine `JUMPDEST`-Anweisung auftritt. Der nachfolgende `PUSH1`-Opcode wird daher mit der Adresse 0 versehen. Dagegen führt der `STOP`-Befehl an der Stelle 136 nicht zu einer Trennung, da die nächste Instruktion ein Sprungziel kennzeichnet.

Diese Regel hat jedoch einige Nachteile. Zum einen könnte der Konstruktorargumente-Bereich, falls der Bytecode einen solchen beinhaltet, in etliche separate Programmteile mit der Größe von einem Byte zerteilt werden. Dies ist der Tatsache geschuldet, dass der Opcode `STOP` mit dem Wert `0x00` kodiert wird. Da die Konstruktorargumente, wie in Abbildung 5.10 gezeigt, diesen Wert oftmals beinhalten, würde diese Heuristik unnötig viele Zerteilungen vornehmen. Ist beispielsweise der Eingangsparameter des Konstruktors ein relativ kleiner Wert vom Typ `uint`, bleiben die restlichen Bytes des Konstruktorarguments mit dem Wert `0` initialisiert, was dazu führt, dass diese Werte als Sequenz von `STOP`-Befehlen interpretiert werden.

Ein weiteres Problem ist, dass diese Heuristik nur für Contracts, die mit einem Solidity-Compiler ab Version 0.4.7 übersetzt wurden, anwendbar ist. Erst ab dieser Version wurde als abschließender Opcode eines separaten Contract-Codes zuerst die Anweisung `STOP` und danach die Instruktion `INVALID` verwendet. Bei früheren Compiler-Versionen ist es

```
000 PUSH1 80
002 PUSH1 40
004 MSTORE
005 CALLVALUE
...
187 CODECOPY
188 PUSH1 00
190 RETURN
191 STOP

000 PUSH1 80
002 PUSH1 40
004 MSTORE
005 PUSH1 04
...
135 JUMPDEST
136 STOP
137 JUMPDEST
138 CALLVALUE
...
```

Abbildung 5.14: Auszug aus dem Output des *Bytecode Splitters* unter Anwendung der Programmende-Heuristik

auch möglich, dass ein ausführbares Programmstück mit einem der Terminierungsbefehle `RETURN`, `REVERT` oder `SELFDESTRUCT` endet.

Aus den genannten Gründen wurde eine alternative Heuristik untersucht, welche die Separation der Programmteile auch für Contracts früherer Solidity-Compiler-Versionen ermöglicht.

Alternative Heuristik zur Trennung des Bytecodes: Programmbeginn bestimmen

Anders als die zuvor beschriebene Heuristik ermittelt dieser Ansatz den Beginn eines ausführbaren Bytecode-Programms. Dafür konnten in der Solidity-Historie bestimmte Sequenzen von EVM-Instruktionen detektiert werden, die den Beginn eines separat auszuführenden Contract-Teils kennzeichnen.

Die betroffenen Opcode-Abfolgen werden in den Abbildungen 5.15 und 5.16 dargestellt. Die zugehörigen hexadezimalen Werte, die dem *Bytecode Splitter* ermöglichen, den gesamten Code in separate Teile zu trennen, sind `0x6060604052` bzw. `0x6080604052`. Hierbei werden zwei Werte auf den Stack gelegt und anschließend der erste an der Stelle des zweiten im Memory gespeichert. Die zugrunde liegende Sequenz entspricht dem *Free Memory Pointer*. Mithilfe dieser Befehlsfolge wird ein Bereich im Memory für bestimmte Zwecke reserviert. Da sich die Adresse des *Free Memory Pointers* und somit der Parameter

```
000 PUSH1 60
002 PUSH1 40
004 MSTORE
```

Abbildung 5.15: Die ursprüngliche Version des *Free Memory Pointers* zur Bestimmung des Programmbeginns

```
000 PUSH1 80
002 PUSH1 40
004 MSTORE
```

Abbildung 5.16: Die aktuelle Version des *Free Memory Pointers* zur Bestimmung des Programmbeginns

der ersten `PUSH1`-Anweisung im Laufe der Zeit verändert haben, sind für den Beginn eines ausführbaren Programmstücks aktuell zwei Bytecode-Sequenzen möglich.

Heuristik zur Entfernung des Metadata-Hashes

In Abschnitt 5.1.6 wurden die in Solidity möglichen Metadata-Hash-Formate angeführt. Der Splitter durchsucht den EVM-Bytecode mithilfe von *Pattern Matching* nach Vorkommnissen dieser Strukturen. Konkret wird dabei nach den regulären Ausdrücken `0xa165` bzw. `0xa265` gesucht, da diese den Beginn des Metadata-Hashes kennzeichnen. Auf Grundlage des Suchergebnisses werden im ersten Fall die letzten beiden Positionen des 43 Bytes langen Wertes mit `0x0029` verglichen und die 43 Bytes aus dem Code entfernt, sofern die Überprüfung erfolgreich verläuft. Beim zweiten Fall wird das selbe Vorgehen angewandt, jedoch mit dem Unterschied, dass der Hash mit der Länge von 52 Bytes entfernt wird, wenn die letzten beiden Hex-Werte `0x0032` entsprechen.

Heuristik zur Erkennung von Creation-Codes

Damit bei der Dekompilierung zwischen dem Konstruktor und den Funktionen unterschieden werden kann, ist es notwendig, im EVM-Bytecode zwischen Creation- und Runtime-Codes zu unterscheiden. Der Creation-Code beinhaltet bekanntlich den Konstruktor, der Runtime-Code die Funktionen eines Contracts. Aus diesem Grund erkennt der *Bytecode Splitter* unter Anwendung der in Abbildung 5.17 dargestellten Befehlsfolge, ob es sich beim Bytecode-Programm um den Creation-Code handelt. Diese Sequenz wird immer zum Abschluss der Erzeugung des Contracts ausgeführt, damit der Runtime-Code zur EVM retourniert und somit zur Ausführung bereit gemacht wird.

```
065 CODECOPY
066 PUSH1 00
068 RETURN
```

Abbildung 5.17: Mögliche Sequenz zur Erkennung eines Creation-Codes

5.3.2 RQ2: Methoden zur Wiederherstellung von Solidity-Code

In diesem Abschnitt wird die Forschungsfrage „Wie kann Solidity-Code aus EVM-Bytecode wiederhergestellt werden?“ bearbeitet. Um diese Frage zu beantworten, werden die Methoden, die zur Solidity-Code-Reproduktion angewandt wurden, angeführt und detailliert beschrieben. Zu diesem Zweck wurde, basierend auf der Entscheidung aus Abschnitt 5.2, auf einem bereits existierenden EVM-Decompiler aufgebaut, um danach den Output sukzessive zu verbessern. Der in der dynamischen Programmiersprache *Python* entwickelte Prototyp mit dem Namen *Soldec* ist also eine Weiterentwicklung von *Erays*.

Aufbauend auf der Beschreibung von Erays aus Abschnitt 4.4 werden in den folgenden Bereichen einige Mängel des Tools hinsichtlich Dekompilierung und Code-Generierung aufgezeigt. Diese Defizite werden umgehend mit Verbesserungsvorschlägen adressiert, welche in Soldec umgesetzt wurden. Ziel dabei ist es, die dekompilierte Ausgabe von Erays in einen Solidity-ähnlichen-Code zu transferieren und dadurch übersichtlicher zu gestalten.

Um die Verbesserung des Outputs von Soldec gegenüber Erays zu veranschaulichen, wird der EVM-Bytecode des Beispiel-Contracts aus Abschnitt 5.1 mit beiden Tools dekompiliert und miteinander verglichen. Siehe dazu Abbildung 5.18 und 5.19 (Erays-Ausgaben) sowie Abbildung 5.20 und 5.21 (Soldec-Outputs). Die Ausgaben beider Tools wurden für die bessere Lesbarkeit in dieser Arbeit dahingehend angepasst, dass einzelne Zeilenumbrüche eingefügt bzw. entfernt wurden. Diese Maßnahme war notwendig, damit die Beispiele auf den jeweiligen A4-Seiten übersichtlich dargestellt werden können.

```
0x89
--------
assert(0 == msg.value)
$s2 = 0x0
$s3 = 0x0
$s4 = 0x0
while (0x1) {
  if ((0xff & $s4) >= s[0x2])
   break
  $s7 = 0xff & $s4
  assert($s7 < s[0x2])
  m[0x0] = 0x2
  if (s[sha3(0x0, 0x20) + $s7] > $s3){
    $s6 = 0xff & $s4
    assert($s6 < s[0x2])
    m[0x0] = 0x2
    $s3 = s[sha3(0x0, 0x20) + $s6]
    $s2 = $s4
  }
  $s4 = 0x1 + $s4
}
m[$m] = 0xff & $s2
return($m, 0x20)
```

Abbildung 5.18: Der dekompilierte Output von Erays: Die Funktion *winningProposal()* des Contracts *Ballot*

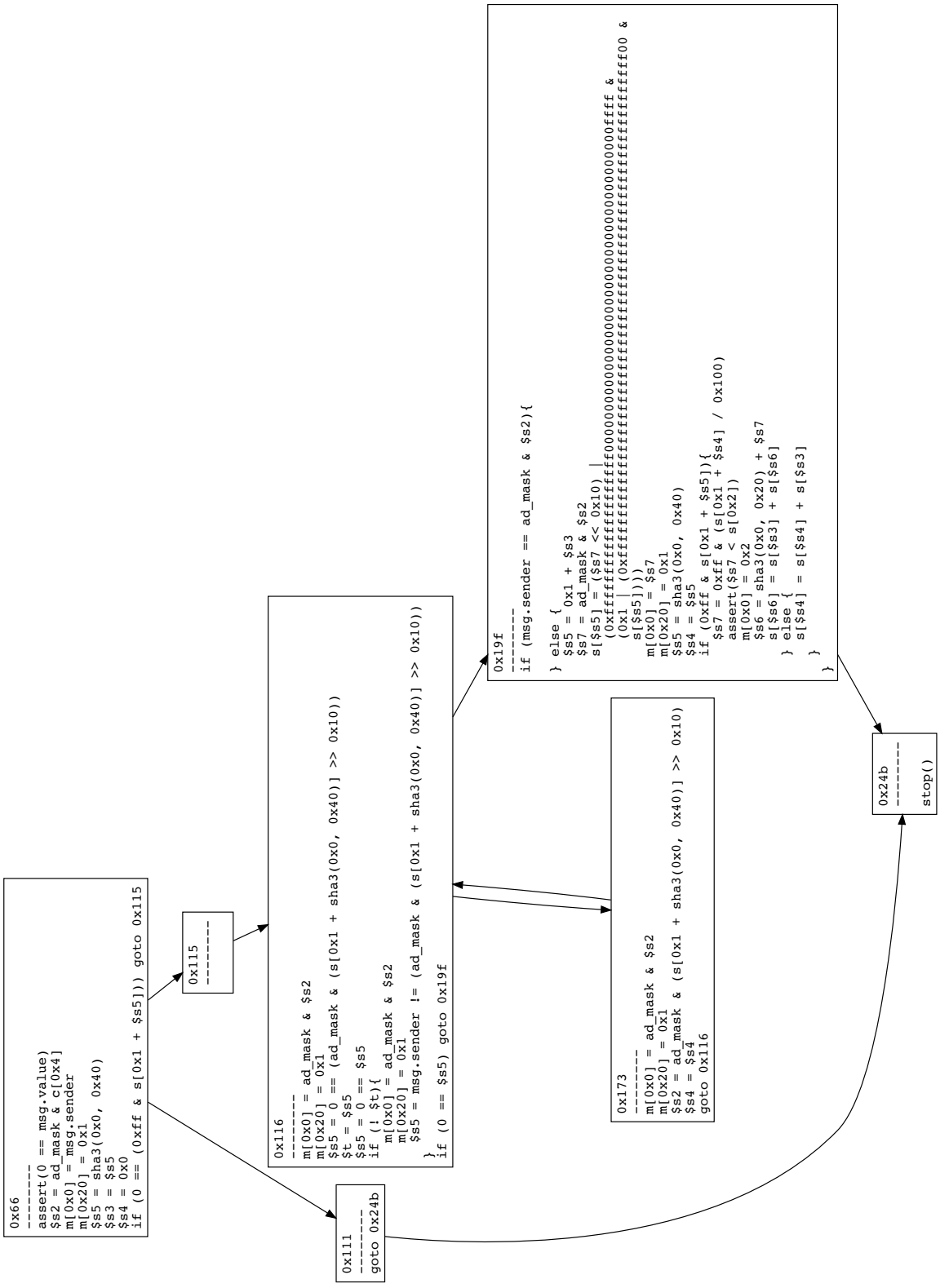

Abbildung 5.19: Der dekompilierte Output von Erays: Die Funktion *delegate(address)* des Contracts *Ballot*

```
constructor() public {
// block 0x0
  require(0 == msg.value);
  codecopy(0x80, 0x487, 0x20);
  unknown $m = 0xa0;
  _storage0 = msg.sender |
      (0xffffffffffffffffffffffff0000000000000000000000000000000000000000 &
      _storage0);
  _storage1[(msg.sender |
      (0xffffffffffffffffffffffff0000000000000000000000000000000000000000 &
      _storage0))] = 0x1;
  uint8 var1 = 0xff & m[0x80];
  unknown var5 = _storage2;
  _storage2 = 0xff & m[0x80]
  if (var5 > 0xff & m[0x80]) {
    uint var6 = sha3r(0x2) + var5;
    unknown var8 = var1 + sha3r(0x2);
    while (true) {
      if (var6 <= var8) {
        break;
      }
      s[var8] = 0x0;
      uint var8 = 0x1 + var8;
    }
  }
  codecopy(0x0, 0xc0, 0x3c7);
  return 0x0;
}

function delegate(address _args1) public {
// block 0x66
  require(0 == msg.value);
  address var2 = _args1;
  unknown var3 = sha3r(msg.sender, 0x1);
  unknown var4 = 0x0;
  if (0xff & _storage1[msg.sender]) {

  } else {
    while (true) {
      if (!((0 != (_storage1[var2] >> 0x10)) &&
            msg.sender != (_storage1[var2] >> 0x10))) {
        break;
      }
      address var2 = (_storage1[var2] >> 0x10);
      unknown var4 = var4;
    }
    if (msg.sender == var2) {

    } else {
      _storage1[var3] = (var2 << 0x10) |
          (0xffffffffffffffffffff0000000000000000000000000000000000000000ffff &
          (0x1 |
          (0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00 &
          _storage1[var3])));
      unknown var4 = sha3r(var2, 0x1);
      if (0xff & _storage1[var2]) {
        assert((0xff & (_storage1[var4] / 0x100)) < _storage2);
        _storage2[0xff & (_storage1[var4] / 0x100)] = s[var3] +
            _storage2[0xff & (_storage1[var4] / 0x100)];
      } else {
        s[var4] = s[var4] + s[var3];
      }
    }
  }
  return;
}
```

Abbildung 5.20: Der dekompilierte Output von Soldec: Der erste Teil des Contracts *Ballot*

```
function winningProposal() public {
// block 0x89
  require(0 == msg.value);
  unknown var2 = 0x0;
  unknown var3 = 0x0;
  unknown var4 = 0x0;
  while (true) {
    if ((0xff & var4) >= _storage2) {
      break;
    }

    assert(0 != ((0xff & var4) < _storage2));
    if (_storage2[0xff & var4] > var3) {
      assert((0xff & var4) < _storage2);
      unknown var3 = _storage2[0xff & var4];
      unknown var2 = var4;
    }
    uint var4 = 0x1 + var4;
  }
  return 0xff & var2;
}

function giveRightToVote(address _args1) public {
// block 0xb4
  require(0 == msg.value);
  address var2 = _args1;
  if (msg.sender != _storage0 || 0xff & _storage1[var2]) {

  } else {
    _storage1[var2] = 0x1;
  }
  return;
}

function vote(uint8 _args1) public {
// block 0xd5
  require(0 == msg.value);
  uint8 var2 = 0xff & _args1;
  unknown var3 = sha3r(msg.sender, 0x1);
  if (0xff & _storage1[msg.sender] || (0xff & var2) >= _storage2) {

  } else {
    _storage1[var3] = (0x100 * (0xff & var2)) |
        (0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00ff &
        (0x1 |
        (0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00 &
        _storage1[var3])));
    assert((0xff & var2) < _storage2);
    _storage2[0xff & var2] = s[var3] + _storage2[0xff & var2];
  }
  return;
}

function() public {
// block 0x61
  revert(0x0, 0x0);
}
```

Abbildung 5.21: Der dekompilierte Output von Soldec: Der zweite Teil des Contracts *Ballot*

Umstellung auf Konsolen-Output

Aus Abbildung 5.18 sowie 5.19 geht hervor, dass Erays seinen dekompilierten Output blockweise als Grafik darstellt. Dabei wird pro Contract-Funktion eine eigene PDF-Datei erstellt, welche mit dem 4-Byte-Hash der Funktion benannt ist. Diese Darstellung trägt allerdings nicht sonderlich zur Übersichtlichkeit bei, zumal der Contract im Gesamten relevant ist. Außerdem ist es nur mit erhöhtem Aufwand möglich, eine bestimmte Funktion ausfindig zu machen und danach zu analysieren, da die Dateien nicht mit aussagekräftigen Funktionssignaturen gekennzeichnet sind. Zudem erhält man im schlimmsten Fall mehrere nicht zusammenhängende bzw. identifizierbare Dateien oder gar überschriebene Files, falls mehrere EVM-Bytecodes unmittelbar nacheinander dekompiliert werden, da die Funktionssignaturen der unterschiedlichen Contracts identisch sein könnten.

Aufgrund der erwähnten Nachteile wurde die Entscheidung getroffen, den Output des Programms so anzupassen, dass dieser eine höhere Ähnlichkeit zu einem Contract der Sprache Solidity aufweist. Dies wird in Soldec durchgeführt, indem die Funktionen des dekompilierten Contracts nicht mehr als separate PDF-Dateien, sondern im sequentiellen Source-Code-Format auf der Konsole ausgegeben werden. Wie in Abbildung 5.20 bzw. 5.21 gezeigt wird, ähnelt diese Quelltext-Ausgabe dem ursprünglichen Solidity-Code und mithilfe von weiteren Optimierungen können auch die Funktionen besser identifiziert und mehrere Contracts gleichzeitig dekompiliert werden. Weitere Informationen dazu befinden sich in den folgenden Abschnitten.

Erkennung von Event- und Funktionssignaturen

Die Funktionssignaturen, welche im EVM-Bytecode mithilfe von 4-Byte-Hashes repräsentiert werden, werden von Erays bereits korrekt aus dem Bytecode herausgelesen. Dieser wird dabei sequenziell traversiert und auf bestimmte Befehlsfolgen durchsucht. Die verwendeten Sequenzen entsprechen dem Funktionsselektor, der die 4-Byte-Hashes der Funktionssignaturen auf den Stack legt, mit dem Hash der aufgerufenen Funktion vergleicht und zum entsprechenden Rumpf springt. Der Funktionsselektor des Ballot-Contracts findet sich in Abbildung 5.22.

Die Befehlsfolgen, die in Erays zur Anwendung kommen, sind `DUP1 PUSH4 EQ PUSHx JUMPI` bzw. `PUSH4 DUP2 EQ PUSHx JUMPI`, wobei die variable Anweisung `PUSHx` in diesem Fall für das Hinzufügen einer beliebigen Sprungadresse auf den Stack steht. Anhand des Arguments der jeweiligen `PUSH4`-Anweisung können die Signaturen extrahiert werden. Im Beispiel aus Abbildung 5.22 sind das die hexadezimalen Werte `5c19a95c`, `609ff1bd`, `9e7b8d61` und `b3f98adc`. Diese 4-Byte-Hashes entsprechen mit einer gewissen Wahrscheinlichkeit der Reihe nach den Funktionssignaturen `delegate(address)`, `winningProposal()`, `giveRightToVote(address)` sowie `vote(uint8)` des Ballot-Contracts.

Die letztgenannte Information ist in Erays allerdings nicht vorhanden. Zur Identifikation der Funktionen werden die 4-Byte-Signaturen verwendet, indem diese als Namen der generierten PDF-Files fungieren. Dadurch könnte man die Routine zumindest manuell

```
013 PUSH4 ffffffff
018 PUSH29 0100000000000000000000000000000000000000000000000000000000
048 PUSH1 00
050 CALLDATALOAD
051 DIV
052 AND
053 PUSH4 5c19a95c
058 DUP2
059 EQ
060 PUSH2 0066
063 JUMPI
064 DUP1
065 PUSH4 609ff1bd
070 EQ
071 PUSH2 0089
074 JUMPI
075 DUP1
076 PUSH4 9e7b8d61
081 EQ
082 PUSH2 00b4
085 JUMPI
086 DUP1
087 PUSH4 b3f98adc
092 EQ
093 PUSH2 00d5
096 JUMPI
```

Abbildung 5.22: Der Funktionsselektor des Contracts *Ballot*

identifizieren, falls man über die benötigten Informationen verfügt. Da dieses Vorgehen allerdings mühsam und fehleranfällig ist, wurde die Funktionsweise in der aktuellen Arbeit adaptiert.

Soldec versucht, anhand des 4-Byte-Hashes auf die tatsächliche Funktionssignatur (bestehend aus Namen und Parametertypen der Funktion) zu gelangen, damit die Routinen entsprechend benannt und somit einfacher identifiziert werden können. Hierzu wurde eine Sammlung von etwa 300.000 Funktionssignaturen mit zugehörigen 4-Byte-Hashes herangezogen. Auch mithilfe der Online-Datenbank *4byte.directory* könnte man die Funktionssignaturen anhand des 4-Byte-Hashes über das entsprechende API abrufen [4].

Ein möglicher Nachteil dieser Technik wurde bereits kurz angerissen, stellt diese nämlich keine vollständig funktionierende Methode zur Identifikation von Funktionssignaturen dar. Dementsprechend ist es nur mit einer bestimmten Wahrscheinlichkeit möglich, vom 4-Byte-Hash auf die ursprüngliche Funktionssignatur zu schließen. Mehr zu diesem Thema findet sich in Abschnitt 5.4, welcher sich mit der Evaluierung des Prototyps befasst.

Ein ähnliches Vorgehen wird für Eventsignaturen verwendet, jedoch mit der Absenz des zuvor erwähnten Mankos. Denn diese werden nicht vom 4-Byte-, sondern vom

tatsächlichen Hash mit der vollen Länge von 32 Bytes im EVM-Bytecode repräsentiert. Auch für diesen Zweck besitzt die TU Wien eine Sammlung, die zum aktuellen Zeitpunkt ungefähr 35.000 Eventsignaturen samt ihrer 32-Byte-Hashes umfasst.

Unterscheidung von Konstruktor und Fallback-Funktion

Erays unterscheidet bei der Dekompilierung nicht zwischen Konstruktor und Fallback-Funktion. Hat die Funktion keine Signatur, was auf die beiden angesprochenen Spezialarten zutrifft, wird der Defaultwert `0xffffffff` zur Darstellung verwendet. Bei der gleichzeitigen Dekompilierung von Creation- und Runtime-Code, was mithilfe des *Bytecode Splitters* ermöglicht wird, würde der dekompilierte PDF-Output der Fallback-Funktion somit die Datei des Konstruktors überschreiben, da diese identisch benannt sind. Mithilfe von „Umstellung auf Konsolen-Output“ wäre zumindest Letzteres gelöst, die Unterscheidung zwischen Konstruktor und Fallback-Funktion ist dennoch nicht mit einem Blick auf die Signatur möglich.

Zu diesem Zweck wurde, wie in Abschnitt 5.3.1 unter „Heuristik zur Erkennung von Creation-Codes“ beschrieben, im *Bytecode Splitter* die Detektion dieses Initialisierungsprogramms implementiert. In Soldec wird diese Information verwendet, um die Signatur des Konstruktors mit dem Wert `0x00000000` zu kennzeichnen, damit in weiterer Folge das Schlüsselwort `constructor` vergeben werden kann. Die Fallback-Funktion wird nach wie vor mit `0xffffffff` markiert, um diese danach wie im ursprünglichen Solidity-Code mit `function()` darstellen zu können. Dies ist mit einem Blick auf die dekompilierte Version des Contracts `Ballot` in Abbildung 5.20 bzw. 5.21 erkennbar. Dass es bei der Dekompilierung des Rumpfes von Fallback-Funktion bzw. Konstruktor noch Forschungsbedarf gibt, wird in Abschnitt 5.4 diskutiert.

Unterscheidung von Securitychecks

Wie im Paper von Erays angeführt, werden die Sicherheitschecks des Solidity-Compilers vom Tool als `assert`-Anweisung dekompiliert, um redundante Blöcke zu entfernen [55]. Tatsächlich existieren diese Absicherungen auch in der höheren Programmiersprache Solidity. Nach dem Kompilieren des Quelltextes zu EVM-Bytecode werden diese Ausdrücke zu bedingten Anweisungen, die im Fehlerfall mit einzelnen `INVALID`-Befehlen, oder wie es in früheren Solidity-Versionen der Fall war, mit einem Sprung an eine ungültige Sprungadresse terminieren. Diese Überprüfungen können dabei vom Entwickler selbst im Solidity-Code untergebracht oder aber vom Compiler nachträglich in den EVM-Bytecode eingefügt werden. Eine Unterscheidung der beiden Arten ist nicht möglich, wie in Abschnitt 5.4 beschrieben wird.

Das weitaus größere Problem an Erays ist jedoch, dass nicht nur `INVALID`-Befehle und Sprünge an ungültige Adressen als `assert` abgebildet werden, sondern auch bedingte Anweisungen, die mit dem Opcode `REVERT` enden. Dieser sorgt im Gegensatz zu `INVALID` dafür, dass Daten und verbleibendes Gas zurückgegeben werden. Eine Unterscheidung dieser beiden Befehle ist somit von essenzieller Bedeutung. In der

```
611 DUP2
612 PUSH1 02
614 DUP3
615 PUSH1 ff
617 AND
618 DUP2
619 SLOAD
620 DUP2
621 LT
622 ISZERO
623 ISZERO
624 PUSH2 0275
627 JUMPI
628 INVALID
```

Abbildung 5.23: Auszug aus dem EVM-Bytecode: Sequenz wird zu einem *Assert*-Check dekompiliert

```
102 JUMPDEST
103 CALLVALUE
104 DUP1
105 ISZERO
106 PUSH2 0072
109 JUMPI
110 PUSH1 00
112 DUP1
113 REVERT
```

Abbildung 5.24: Auszug aus dem EVM-Bytecode: Sequenz wird zu einem *Require*-Check (dem *Non-Payable-Check* des Compilers) dekompiliert

EVM sind die Operationscodes `INVALID` bzw. `REVERT` somit quasi Pendants zu den Solidity-Anweisungen `assert` bzw. `require`. Laut der Solidity-Dokumentation wird Erstere lediglich dafür verwendet, um auf interne Fehler und Invarianten zu testen. Da es sich hierbei um unerwartetes Verhalten handelt, ist es legitim, dass das eingesetzte Gas nicht mehr retourniert wird. Im Gegensatz dazu wird `require` verwendet, um zu prüfen, ob die für die Ausführung eines bestimmten Programmstücks notwendigen Bedingungen erfüllt sind [27]. Als Beispiel fungiert die Voraussetzung von bestimmten Werten als Eingangs- bzw. Ausgangsparameter oder Zustandsvariablen des Contracts. Da es sich bei diesen Checks um korrekte Abläufe im Programm handelt, werden die bis zu dem Zeitpunkt durchgeführten Änderungen zwar zurückgesetzt, das verbleibende Gas allerdings wieder retourniert. Auch diese Art der Überprüfung kann entweder vom Entwickler selbst im Solidity-Code platziert oder nachträglich vom Solidity-Compiler eingefügt werden. Ein konkretes Beispiel stellt der *Non-Payable-Check* dar, welcher vom Compiler durchgeführt wird, um zu garantieren, dass einer Funktion, welche nicht mit dem Schlüsselwort `payable` markiert ist, kein ETH-Wert übertragen wird. Dieser Check

stellt die erste Anweisung einer Funktion dar, wie man in der dekompilierten Ausgabe (Abbildung 5.20 bzw. 5.21) erkennen kann.

Da die Unterscheidung zwischen `assert` und `require` aus den obigen Gründen eine wesentliche darstellt, wurde diese als Optimierung im Prototyp Soldec umgesetzt. `INVALID`-Opcodes sind weiterhin ein Anzeichen für `assert`-Befehle, `REVERT`-Anweisungen werden in der dekompilierten Ausgabe allerdings als `require`-Instruktionen dargestellt. Eine mögliche Bytecode-Sequenz, welche zu einer `assert`-Anweisung dekompiliert wird, findet sich in Abbildung 5.23. Dagegen wird die Befehlsfolge aus Abbildung 5.24 in eine `require`-Instruktion übersetzt, welche in diesem speziellen Fall den *Non-Payable-Check* und somit die erste Anweisung einer Funktion darstellt.

Umschreibung von bestimmten Memory-Aktionen

Bei Erays werden die Memory-Aktionen `MLOAD`, `MSTORE` und `MSTORE8` immer als direkte Zugriffe auf den volatilen Speicher dekompiliert. Wie in Abbildung 5.18 ersichtlich wird ein bestimmter Memory-Bereich angesprochen, indem das Kürzel `m` gefolgt von einer Adresse in eckigen Klammern angeführt wird. `m[0x0]` steht also beispielsweise für das Memory an der Adresse `0`.

Solche Zugriffe sind jedoch in den meisten Fällen nicht im ursprünglichen Solidity-Code enthalten und wurden folglich nachträglich vom Compiler in den EVM-Bytecode eingefügt. Grund für dieses Verhalten sind höchstwahrscheinlich nachfolgende Befehle, die ihre Argumente aus dem Memory beziehen und im EVM-Bytecode deshalb unmittelbar nach den betroffenen Memory-Speicherungen stattfinden. Fachlich gehören diese Zugriffe auf den flüchtigen Speicher also zu den anschließenden Instruktionen.

In dieser Arbeit wurde deshalb versucht, die korrespondierenden Operationen für die EVM-Befehle `SHA3`, `LOG0` bis `LOG4` sowie `RETURN` zusammenzufassen, um eine höhere Ähnlichkeit zum ursprünglichen Solidity-Code zu erzielen. In diesem werden nur explizite Memory-Aktionen, wie z. B. das Schreiben in eine Memory-Variable dargestellt, nicht jedoch die vorbereitenden Tätigkeiten für die genannten Opcodes.

In Abbildung 5.25 ist eine Befehlssequenz der EVM dargestellt, welche für das Erstellen eines SHA3-Hashes verantwortlich ist. Dabei wird zuerst die Adresse des Nachrichtensenders (`CALLER`) im Memory an der Position `0` gespeichert. Zusätzlich wird der Wert `1` in den volatilen Speicher hinter die Adresse des Aufrufers geschrieben. Die Operation `SHA3` greift anschließend auf die zuvor beschriebenen Memory-Slots zu und berechnet von der Konkatenation der beiden Werte einen Keccak-256-Hash. In der dekompilierten Ausgabe wird diese Befehlsfolge vorerst zur Instruktion `sha3r(msg.sender, 0x1)` zusammengefasst, welche noch keiner gültigen Solidity-Anweisung entspricht. Diese Vorgehensweise wird nämlich wiederum als vorbereitende Tätigkeit verwendet, um auf einen bestimmten Wert in einem Mapping zuzugreifen. Da Mappings üblicherweise nicht im Memory, sondern im Storage gespeichert werden, ist hier ein weiterer Dekompilierungsschritt notwendig, welcher im nächsten Abschnitt mit dem Titel „Umschreibung von Storage-Aktionen“ beschrieben wird.

```
240 JUMPDEST
241 CALLER
242 PUSH1 00
244 SWAP1
245 DUP2
246 MSTORE
247 PUSH1 01
249 PUSH1 20
251 DUP2
252 SWAP1
253 MSTORE
254 PUSH1 40
256 DUP3
257 SHA3
```

Abbildung 5.25: Auszug aus dem EVM-Bytecode: Sequenz zum Bilden eines SHA3-Hashes

Die Befehle zur Signalisierung von Events werden auf ähnliche Art und Weise umgeschrieben. Mithilfe dieser Technik können die `LOG`-Opcodes in den Solidity-Befehl `emit` übersetzt werden, ohne die vorbereitenden Instruktionen zum Beschreiben der betroffenen Memory-Positionen ausgeben zu müssen. Unter Zuhilfenahme der Sammlung von Eventsignaturen kann, wie im Abschnitt „Erkennung von Event- und Funktionssignaturen“ beschrieben, auch die Signatur des auszugebenden Events ermittelt werden.

In Abbildung 5.26 findet sich der von Soldec dekompilierte Output einer modifizierten Version der Funktion `transfer(address,uint256)` des Contracts `BasicToken` aus Abbildung 5.2. Die Unterschiede der hier verwendeten Version liegen darin, dass zusätzlich zur durchgeführten Funktionalität, bei welcher der Sender der Nachricht einen bestimmten Betrag an den Empfänger transferiert, diese Tätigkeit über ein Event geloggt wird. In der dekompilierten Ausgabe ist in der Zeile vor dem Rückgabewert zu erkennen, dass das Emittieren des Events `Transfer(address,address,uint256)` wiederhergestellt werden konnte. Im Gegensatz dazu werden in Abbildung 5.27, welche den von Erays dekompilierten Output derselben Funktion zeigt, die zahlreichen Memory-Aktionen explizit dargestellt, was eine Fülle von Befehlen verursacht, die nicht dem ursprünglichen Solidity-Code entsprechen.

Auch der Opcode `RETURN` bezieht seine Argumente aus dem Memory. Wenn man die dekompilierten Ausgaben der Funktion `winningProposal()` beider Tools miteinander vergleicht (Abbildung 5.18 bzw. 5.21), erkennt man, dass die Beschreibung der Memory-Position, welche danach vom Befehl `return` gelesen wird, eingespart werden kann.

Der ursprüngliche Code:

```
m[$m] = 0xff & $s2
return($m, 0x20)
```

wird somit zu:

```
function transfer(address _args1, uint256 _args2) public {
// block 0x43f
  require(0 == msg.value);
  require(_storage1[msg.sender] >= _args2);
  _storage1[msg.sender] = _storage1[msg.sender] - _args2;
  _storage1[_args1] = _storage1[_args1] + _args2;
  emit Transfer(/*address*/ msg.sender , /*address*/ _args1,
      /*uint256*/ _args2);
  return true;
}
```

Abbildung 5.26: Der dekompilierte Output von Soldec: Eine alternative Version der Funktion *transfer(address,uint256)* eines Token-Contracts

```
0x43f
--------
assert(0 == msg.value)
$s4 = ad_mask & c[0x4]
$s3 = c[0x24]
m[0x0] = msg.sender
m[0x20] = 0x1
assert(0 == (s[sha3(0x0, 0x40)] < $s3))
m[0x0] = msg.sender
m[0x20] = 0x1
$s6 = sha3(0x0, 0x40)
s[$s6] = s[$s6] - $s3
m[0x0] = ad_mask & $s4
m[0x20] = 0x1
$s6 = sha3(0x0, 0x40)
s[$s6] = s[$s6] + $s3
m[$m] = $s3
log3($m, (0x20 + $m) - $m,
  0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef,
  msg.sender, ad_mask & $s4)
m[$m] = 0x1
return($m, (0x20 + $m) - $m)
```

Abbildung 5.27: Der dekompilierte Output von Erays: Eine alternative Version der Funktion *transfer(address,uint256)* eines Token-Contracts

```
return 0xff & var2;
```

Umschreibung von Storage-Aktionen

In Solidity werden Contract-Variablen im Storage gespeichert. In der Ausführungsumgebung und so auch bei der Übersetzung in EVM-Bytecode ist zu beachten, dass diese Art von Variablen beginnend an der Storage-Adresse `0` positioniert werden. Im Contract in Abbildung 5.2 wird die Variable `totalSupply_` folglich an der Storage-Position `0`, das Mapping `balances` an der Stelle `1` gespeichert. Dieses Verhalten spiegelt sich bei der Dekompilierung wider. Da die Variablennamen nicht reproduzierbar sind, wie in Abschnitt 5.4 diskutiert wird, wird in Soldec die Notierung `_storage` in Kombination

mit der Position verwendet. In einem konkreten Beispiel (Funktion `totalSupply()` des Contracts `BasicToken`), welches den Zugriff auf die Variable `totalSupply_` darstellt, wird die Erays-Schreibweise `s[0x0]` von Soldec durch `_storage0` ersetzt, damit eine höhere Ähnlichkeit zum ursprünglichen Solidity-Code erzielt werden kann. Die vergleichsweisen Ausgaben beider Tools befinden sich in Abbildung 5.28 bzw. 5.29.

```
function totalSupply() public {
// block 0x5b
  require(0 == msg.value);
  unknown var2 = _storage0;
  return var2;
}
```

Abbildung 5.28: Der dekompilierte Output von Soldec: Funktion *totalSupply()* des Contracts *BasicToken*

```
0x5b
--------
assert(0 == msg.value)
$s2 = s[0x0]
m[$m] = $s2
return($m, 0x20)
```

Abbildung 5.29: Der dekompilierte Output von Erays: Funktion *totalSupply()* des Contracts *BasicToken*

Des Weiteren muss, wie im Abschnitt „Umschreibung von bestimmten Memory-Aktionen“ angeschnitten, der Zugriff auf ein Mapping umgeschrieben werden. Laut der Dokumentation von Solidity wird ein solches immer im Storage gespeichert. Der Zugriff auf ein Mapping erfolgt, indem der Keccak-256-Hash der Konkatenation von Schlüssel und Storage-Position berechnet und anschließend von `SLOAD` bzw. `SSTORE` als Adresse verwendet wird [27]. Mit dieser Herangehensweise wird beispielsweise die im letzten Abschnitt determinierte Anweisung `sha3r(msg.sender, 0x1)` bei einem nachfolgenden Storage-Opcode zu einem Zugriff auf das Mapping an der Storage-Position `1` mit dem Key `msg.sender` transformiert.

Das Ergebnis dieser in Soldec angewandten Methodik ist in Abbildung 5.26 ersichtlich. In diesem reproduzierten Quelltext wird mithilfe der Schlüssel `msg.sender` und `_args1` korrekt aus dem Mapping mit dem Titel `_storage1` gelesen bzw. in dieses geschrieben.

Erkennung von Unstrukturierten Bedingungen

Zur Erkennung von bedingten Anweisungen führt Erays eine strukturelle Analyse durch. Dabei wird der CFG traversiert und mithilfe von *Pattern Matching* auf bekannte Kontrollstruktur-Muster durchsucht und gegebenenfalls die entsprechenden Regionen

innerhalb des CFGs zu den abstrakten Konstrukten reduziert. Dieses Vorgehen wurde bereits ausführlich in Abschnitt 3.6 beschrieben. Für die Ermittlung der Anweisungen `if-then` sowie `if-then-else` verwendet Erays die in Abbildung 3.5 dargestellten Patterns.

```
0xb4
--------
assert(0 == msg.value)
$s2 = ad_mask & c[0x4]
$s3 = msg.sender != (ad_mask & s[0x0])
if (! $s3){
  m[0x0] = ad_mask & $s2
  m[0x20] = 0x1
  $s3 = 0xff & s[0x1 + sha3(0x0, 0x40)]
}
if ($s3){

} else {
  m[0x0] = ad_mask & $s2
  m[0x20] = 0x1
  s[sha3(0x0, 0x40)] = 0x1
}

stop()
```

Abbildung 5.30: Der dekompilierte Output von Erays: Funktion *giveRightToVote(address)* des Contracts *Ballot*

Unstructured Conditions (zu Deutsch etwa unstrukturierte bzw. zusammengesetzte Bedingungen) sind bedingte Anweisungen, die sich aus mehr als einer Bedingung zusammensetzen. Die Konditionen werden dabei mithilfe von logischen Operatoren, also dem logischen UND bzw. dem logischen ODER, verknüpft. Die Funktion `giveRightToVote(address)` des Contracts `Ballot` beinhaltet eine solche `if`-Anweisung. In Abbildung 5.30 wird die von Erays erstellte, dekompilierte Ausgabe der genannten Funktion gezeigt. Dabei ist zu beachten, dass die ursprüngliche `if`-Instruktion, bestehend aus einer zusammengesetzten Bedingung, in zwei aufeinanderfolgende `if`-Statements aufgeteilt wird. Die erste bedingte Anweisung ist dabei nur dazu da, die Parameter für den zweiten Check in die entsprechende Form zu bringen. Dieses Vorgehen ist zwar semantisch korrekt und äquivalent, syntaktisch allerdings unterschiedlich, was zu einem unübersichtlicheren Output führt.

Um auch die zusammengesetzten Bedingungen via Pattern Matching als solche detektieren zu können, ist es notwendig, ein neues Muster einzuführen, so wie es Erays bisher für die Kontrollstrukturen `IfThen`, `IfThenElse`, `Sequence` und `Loop` macht. Soldec verwendet also zusätzlich die Struktur `IfCombined`, um auch die unstrukturierten Bedingungen erkennen zu können. Das verwendete Muster ist in Abbildung 5.31 dargestellt.

Die Blöcke `a0`, `a1`, `a2` werden zum Konstrukt `IfCombined` reduziert. Dabei ist zu beachten, dass im Unterschied zur traditionellen `if`-Instruktion sowohl `a0` als auch `a2` eine bedingte Anweisung, also eine Verzweigung (Opcode `JUMPI`) beinhalten. Ob nach

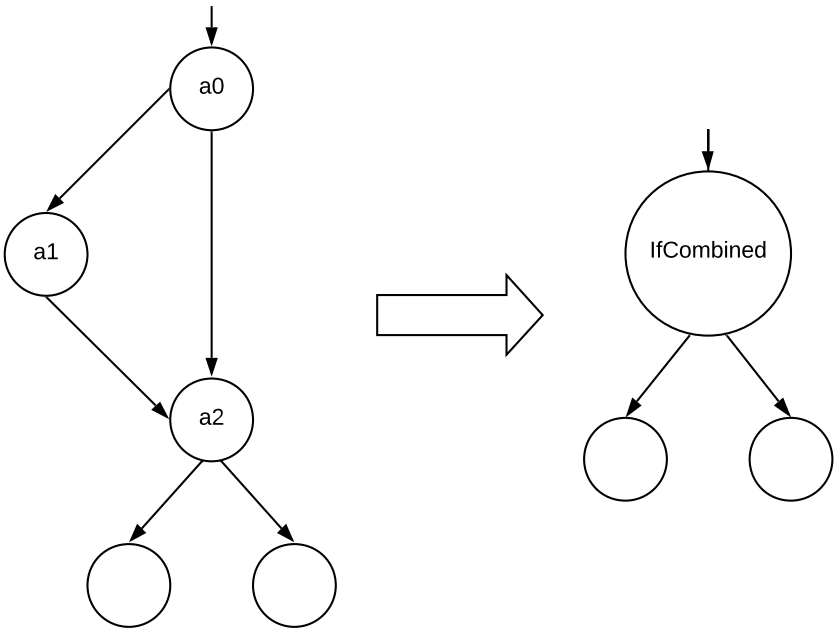

Abbildung 5.31: Pattern Matching für unstrukturierte bzw. zusammengesetzte Bedingungen

```
753 JUMPDEST
754 ISZERO
755 PUSH2 02fb
758 JUMPI
```

Abbildung 5.32: Auszug aus dem EVM-Bytecode: mögliche Sequenz zur Erkennung von unstrukturierten Bedingungen

dem Block `a2` nun tatsächlich eine bedingte Anweisung oder gar eine Schleife platziert ist, ist nicht von Relevanz. `IfCombined` kann als Header all dieser Kontrollstrukturen fungieren.

Eine wesentliche Einschränkung ist allerdings noch zu beachten. Die beiden Bedingungen, die miteinander verknüpft werden sollen, sind in den Blöcken `a0` und `a1` enthalten. Der Block `a2` ist nur dazu da, die Bedingungen in die korrekte Form zu bringen, um diese miteinander verknüpfen zu können. Dementsprechend sind hier neben den strukturellen Checks zusätzlich Bytecode-Überprüfungen vonnöten. Block `a2` darf nach einer heuristischen Approximation nicht aus mehr als sechs Bytes bestehen und muss wie bereits erwähnt mit dem Opcode `JUMPI` enden. Dies ist darauf zurück zu führen, dass der Block keinen eigenen Check durchführen, sondern lediglich die Überprüfungen seiner Vorgänger-Blöcke verwenden darf. In Abbildung 5.32 findet sich eine der möglichen Bytecode-Sequenzen.

Erkennung von Schleifen

Auch die Erkennung von Schleifen wurde in Erays bereits mit dem in Kapitel 3.6 beschriebenen Ansatz *Pattern Matching* umgesetzt. Dabei wird der in Abbildung 3.4 dargestellte *Pre-tested loop* als `while`-Schleife erkannt. Ein positiv dekompiliertes Beispiel dazu findet sich in Abbildung 5.18, welche die Funktion `winningProposal()` des Ballot-Contracts zeigt.

Diese Art der Detektion funktioniert in Erays allerdings nicht für alle Schleifen. Wie die Abbildung 5.19 zeigt, kann in der Funktion `delegate(address)` die entsprechende Region nicht zu einer `while`-Schleife reduziert werden und es verbleiben die ungewollten Sprünge zwischen den einzelnen Blöcken im dekompilierten Programm. Damit solche und auch *Post-tested loops*, also `do-while`-Schleifen, erkannt werden können, wird in Soldec zusätzlich zur strukturellen Analyse das in Abschnitt 3.6.1 erläuterte Konzept *Domination* verwendet. Mithilfe dieser Technik können weitere essenzielle Informationen bezüglich Schleifen gewonnen werden, wie zum Beispiel *Header* und *Tail* sowie die Austrittspunkte und Nachfolger einer Schleife. Diese Daten werden in Soldec schließlich für die genauere Erkennung von Schleifen verwendet. Die Absenz von `for`-Schleifen wird in Abschnitt 5.4 diskutiert.

Wie man anhand von Abbildung 5.20 sieht, trägt diese Technik dazu bei, dass auch die `while`-Schleife der Funktion `delegate(address)` erkannt werden kann und dadurch im gesamten Contract keine Sprünge verbleiben.

Erkennung von vorzeitigen Schleifenausstiegen

Auch die Erkennung von `break`-Anweisungen wird in Soldec unter anderem mithilfe des Dominator-Prinzips umgesetzt. Zusätzlich kann der erstellte Prototyp unter Verwendung der in Abschnitt 3.6.1 angeführten Technik *Loop Successor Refinement* im Vergleich zu Erays außerdem mehrere vorzeitige Schleifen-Austritte erkennen.

Die Anwendung der Konzepte *Domination* und *Back Edge* führt in der Regel nämlich dazu, dass vorzeitige Austritte, also `break`-Anweisungen, als Nachfolger der Schleife erkannt werden. Mithilfe von *Loop Successor Refinement* wird daraufhin für jeden Schleifen-Nachfolger iterativ geprüft, ob dieser vom Loop-Header dominiert wird und ob alle unmittelbaren Vorgänger-Knoten des Nachfolgers bereits zur Schleife gehören. Wenn diese Vermutung zutrifft, wird auch der vermeintliche Nachfolger als Schleifenknoten registriert. Dieses Verfahren wird solange durchgeführt, bis die Schleife nur mehr exakt einen Nachfolger hat. Alle ursprünglichen Nachfolger, welche danach Schleifenknoten darstellen, enthalten zusätzlich eine `break`-Anweisung und verweisen auf den verbleibenden, tatsächlichen Nachfolger der Schleife, sodass diese nur mehr einen einzigen Austrittspunkt besitzt.

Bei der Code-Generierung wird diese Konstruktion als Schleife mit mehreren `break`-Instruktionen dargestellt, wobei diese im Normalfall innerhalb von `if`-Anweisungen auftreten, da der vorzeitige Austritt üblicherweise von einer Bedingung abhängig ist. Die

Einschränkung der verwendeten Implementierung ist, dass auch die Schleifenbedingung, welche üblicherweise z. B. in der `while`-Operation geprüft wird, auch als bedingte Anweisung mit abschließendem `break` dargestellt wird. Dieses Manko ist auch in der von Soldec dekompilierten Version des Ballot-Contracts in Abbildung 5.20 erkennbar und wird in Abschnitt 5.4 adressiert.

Verbesserung der Lesbarkeit von Variablen

Um die generelle Lesbarkeit des dekompilierten Outputs zu verbessern und eine höhere Ähnlichkeit zum ursprünglichen Solidity-Code zu erzielen, wurde zuerst die Benennung der Variablen angepasst. Dabei wurde das in Erays verwendete Präfix `$s` auf `var` geändert. Variablen werden in Soldec somit entsprechend dem chronologischen Vorkommen im Code mit `var1`, `var2`, etc. statt mit `$s1`, `$s2`, etc. benannt.

Des Weiteren werden auch Namen für Funktionsargumente vergeben. Hierfür wird das Präfix `_args` verwendet. In Erays wird beim Lesen eines Parameters die entsprechende Adresse des Calldata-Bereichs im dekompilierten Code angegeben. Dies ergibt unter der Annahme, dass die Länge eines Parameters 32 Bytes einnimmt und der erste Parameter nach dem 4-Byte-Hash auftritt, den Erays-Ausdruck `c[0x4]` für den ersten, `c[0x24]` für den zweiten und so weiter. In Soldec entsprechen diese Werte nach dem erwähnten Prinzip den Parametern `_args1` bzw. `_args2`.

Die abschließend durchgeführte Verbesserung hinsichtlich Lesbarkeit von Variablen wurde mithilfe der in Abschnitt 3.5 beschriebenen Konzepte zur Datenflussanalyse umgesetzt. Erays verwendet das Konzept *Copy Propagation* bereits, indem der zugewiesene Wert einer Variable, auf welche nur ein einziges Mal und nur im selben Block der Variablendeklaration zugegriffen wird, auf den zugreifenden Ausdruck propagiert wird. Unter Anwendung von *Dead Code Elimination* wird die nicht mehr benötigte Variable anschließend entfernt. In Soldec wurde anhand der beiden Techniken zusätzlich eingeführt, dass der Wert einer Variable, welche nur in einem einzigen Block in Verwendung ist, unabhängig der Zugriffsanzahl auf die Zugriffe propagiert wird, um die Variable anschließend wegoptimieren zu können. Alle Zugriffe auf die Variable innerhalb eines Blocks werden also direkt durch ihren Wert ersetzt. In Abschnitt 5.4 wird eine zusätzliche Verbesserung, die blockübergreifende Propagierung der Variablenwerte, diskutiert.

Typerkennung

Bei der Typerkennung herrscht weiterhin enormes Verbesserungspotenzial. Während Erays überhaupt keine Reproduktion von Typen durchführt, wurde in Soldec eine rudimentäre Entwicklung mithilfe eines trivialen Ansatzes gewählt. Dabei erhält eine Variable den Typ `bool` bzw. `uint`, wenn es sich beim Ausdruck, dessen Wert auf die betroffene Variable zugewiesen wird, um eine vergleichende bzw. arithmetische Operation handelt. So wird bspw. dem Ergebnis eines `EQ`-Befehls der Typ `bool` und der Summe einer `ADD`-Anweisung der Typ `uint` zugewiesen.

Einem Ausdruck wird zudem automatisch der Typ `address` zugeteilt, wenn dieser die Adressmaske oder den Nachrichtensender beinhaltet. Nach demselben Prinzip werden die Typen `uint8` bzw. `uint256` vergeben, sofern ein Parameter einer Anweisung die entsprechende Maske repräsentiert.

Ferner konnten einige Typinformationen aus der im Abschnitt „Erkennung von Event- und Funktionssignaturen“ erwähnten Reproduktion der Event- bzw. Funktionssignaturen entnommen werden. Die Weiterverwendung mithilfe einer Typ-Propagierung wurde in dieser Arbeit jedoch nicht umgesetzt. Diese zusätzliche Optimierung wird im folgenden Abschnitt diskutiert.

5.4 Evaluierung des Prototyps

Zum Abschluss des Kapitels wird der im Rahmen dieser Arbeit erstellte Prototyp *Soldec* anhand bestimmter Spracheigenschaften von Solidity evaluiert. Im Zuge der Beantwortung der Forschungsfrage „Wie viel vom ursprünglichen Solidity-Code ist mit den angewandten Techniken wiederherstellbar?“ wird diskutiert, welche Strukturen mit den verwendeten Methoden reproduziert werden können und bei welchen es noch weiterer Forschung bedarf. Diese Evaluierung wird unter Anwendung des Decompilers auf den in Abschnitt 5.1 vorgestellten Contract *Ballot* durchgeführt. Dabei ist noch einmal ein detaillierter Blick auf den dekompilierten Output von Soldec notwendig, welcher in Abbildung 5.20 bzw. 5.21 dargestellt wird.

5.4.1 RQ3: Evaluierung der Wiederherstellung von Solidity-Code

In den vorherigen Abschnitten wurde bereits im Detail diskutiert, wie man bestimmte Solidity-Sprachelemente wiederherstellen kann. Dabei wurde auch ersichtlich, welche Anweisungen bzw. Kontrollstrukturen grundsätzlich mit den verwendeten Methoden reproduziert werden können. Zusammengefasst handelt es sich um folgende:

- Contracts
- Konstruktoren
- Funktionen
- Sicherheitschecks (`require` und `assert`)
- bedingte Anweisungen (`if` und `if-else`)
- Schleifen (`while` und `do-while`)
- vorzeitige Schleifenaustritte (`break`)
- Events

- Storage-Variablen
- lokale Variablen
- Mappings
- Rückgabe-Werte
- Typen von Variablen

Im Laufe von Abschnitt 5.3.2 wurden jedoch Einschränkungen angeschnitten. Einige Sprachelemente konnten nicht zur Gänze wiederhergestellt werden, was jedoch vermutlich mit weiterer Forschung möglich wäre.

Bei der Erkennung von Funktionen ist nur die Reproduktion von öffentlichen (`public` bzw. `external`), nicht jedoch die Wiederherstellung von privaten bzw. internen Funktionen garantiert. Letztere können nur unter Anwendung einer Heuristik angenähert werden, wie es bereits die Tools Erays [55], Vandal [18] oder DSol [34] versuchen. Dass diese Ansätze zuverlässig sind, ist jedoch eher zu bezweifeln. Bei den verschiedenen Tests der genannten Programme, die im Rahmen dieser Arbeit durchgeführt wurden, konnten zahlreiche private Funktionen nicht als solche erkannt werden. Andererseits wurden von den Heuristiken fälschlicherweise private Funktionen erzeugt, obwohl im ursprünglichen Solidity-Code keine enthalten waren.

Bezüglich der Signaturen von öffentlichen Funktionen und Events wurde bereits erwähnt, dass diese aus einer Sammlung von Hash-Werten zu Signaturen entnommen werden. Da es jedoch keine vollständige Sammlung von Signaturen geben kann, weil diese beliebig benannt werden können, ist es nicht unwahrscheinlich, dass zahlreiche Signaturen nicht wiederhergestellt werden können. Um eine möglichst hohe Abdeckung zu erreichen, ist es praktikabel, die *Verified Contracts* von *Etherscan* [29] regelmäßig auf neue Signaturen zu durchforsten und damit das Mapping zu erweitern. Für Funktionen ist es außerdem nicht auszuschließen, dass einige Signaturen denselben 4-Byte-Hash besitzen. Dies ist der Tatsache geschuldet, dass nur die ersten vier Bytes des 32-Byte-Hash zur Identifikation verwendet werden. Die Wahrscheinlichkeit ist zwar relativ gering, aufgrund der begrenzten Anzahl von knapp 4,3 Mrd. möglichen 4-Byte-Hashes jedoch nicht ausgeschlossen. Mit einem derart kleinen Risiko kann man allerdings wahrscheinlich leben.

Bereits angesprochen wurde auch der weitere Forschungsbedarf bei Konstruktoren und Fallback-Funktionen. Bei ersteren ist der Bedarf deutlich höher, da im Creation-Code üblicherweise zusätzliche Arbeiten durchgeführt werden, die im Konstruktor des Solidity-Codes nicht implementiert wurden. In Abbildung 5.20 kann man beispielsweise zahlreiche `codecopy`-Befehle sowie die Initialisierung des Storages in einer Schleife erkennen. Diese Tätigkeiten wurden erst zur Übersetzungszeit vom Solidity-Compiler eingefügt. Hier müsste herausgefunden werden, welche Operationen bereits im ursprünglichen Konstruktor vorhanden waren. Die Fallback-Funktion kann im Normalfall erkannt werden, jedoch müsste sie im konkreten Fall des Ballot-Contracts (Abbildung 5.21) ausgeblendet oder

mit einem kompilierbaren `revert`-Befehl versehen werden, da diese keine Funktionalität enthält und deshalb nicht im ursprünglichen Code vorhanden ist.

In Soldec wird man wie erwähnt mit der Absenz von `for`-Schleifen konfrontiert. Ein weiteres Manko hinsichtlich Darstellung ist, dass die Schleifenbedingungen nicht im Header, sondern in einem `if`-Befehl samt `break`-Anweisung geprüft werden. In der Funktion `winningProposal()` in Abbildung 5.21 sind beide Problematiken zu erkennen. Bei bedingten Anweisungen gibt es eine ähnliche Optimierungsmöglichkeit. Eine `if`-Instruktion in einem `else`-Befehl könnte eventuell zur Anweisung `else if` transformiert werden. Außerdem könnten vorzeitige `return`-Befehle in den entsprechenden Zweig hochgezogen und dadurch die `else`-Anweisung eliminiert werden, wie man beispielsweise am dekompilierten Output der Funktion `giveRightToVote(address)` erkennen kann. All diese syntaktischen Verbesserungen könnten mit *Post-Structuring Optimizations* durchgeführt werden, welche von Yakdan et al. im Schriftstück über den Decompiler Dream beschrieben werden [54].

Was für `break`-Anweisungen bereits in dieser Arbeit umgesetzt wurde, ist auch für die Schlüsselwörter `continue` und `return` notwendig. Tauchen diese Anweisungen innerhalb einer Schleife auf, würden in Soldec wahrscheinlich noch ungewollte `goto`-Befehle verbleiben. Ein möglicher Ansatz zur Lösung dieses Problems sind die *Semantics-Preserving Control-Flow Transformations*, welche wiederum im Paper von Dream erläutert wurden [54].

Bei der Erkennung von lokalen Variablen ist es notwendig, diese nicht nur innerhalb eines Blocks, sondern über eine gesamte Funktion, also blockübergreifend zu propagieren. Dadurch könnten in der Funktion `delegate(address)` in Abbildung 5.20 auch die weiteren Storage-Zugriffe, wie zum Beispiel `s[var4]` bzw. `s[var3]`, korrekt abgebildet werden, da der Inhalt der verwendeten Variablen propagiert und die Variablen danach eliminiert werden könnten. Dasselbe gilt z. B. auch für die Variable `var2` in Abbildung 5.28, welche mithilfe von blockübergreifender Propagierung eingespart werden könnte.

Die Detektion von *Structs* wurde im Gegensatz zu *Mappings* überhaupt nicht implementiert. Damit auch diese reproduziert werden können, bedarf es einer detaillierten Analyse der im dekompilierten Code verbleibenden Masken. Da die Datenstruktur `struct` mehrere Datentypen kapselt und für die Speicherung immer ein neuer Slot im Storage verwendet wird, in welchem die Elemente nacheinander gespeichert werden [27], ist die Maske `ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00` beispielsweise ein Anzeichen dafür, dass an der letzten Stelle eine Variable mit einem Datentyp, welcher einen Platzbedarf von einem Byte hat (also z. B. `uint8` oder `bool`), steht.

Für die Typ-Erkennung wurde lediglich ein naiver Ansatz gewählt. Gänzlich außen vor gelassen wurde die Propagierung von Typen. Dies gilt für lokale, Storage-Variablen genauso wie für die Typen von Funktionen, also für die Typen der Rückgabewerte einer Funktion. Auch für diese Tätigkeit gibt es eine generelle Literatur in der Dekompilierung [38] [3].

Dass weitere Umschreibungen für bestimmte Opcodes notwendig sind, wurde schon bei der Evaluierung von Konstruktor bzw. Fallback-Funktion ersichtlich. Zahlreiche Opcodes, welche in den Beispielen nicht vorkommen, wurden in dieser Arbeit nicht berücksichtigt. Auch diese Befehle beziehen ihre Daten teilwiese aus dem Memory und müssen wie die behandelten Opcodes umgeschrieben werden. Das sind beispielsweise die Befehle zum Aufrufen von externen Contracts (z. B. `CALL`, `CALLCODE` bzw. `DELEGATECALL`) oder zum Erzeugen von weiteren Contracts (z. B. `CREATE`). Eine detaillierte Liste aller EVM-Opcodes ist im *Ethereum Yellow Paper* zu finden [53].

Auch die verwendete Version des Solidity-Compilers, welche mithilfe des Befehls `pragma solidity` als erster Eintrag im Contract fungiert, wurde in dieser Arbeit nicht berücksichtigt. Für Contracts, welche mit einer Compiler-Version vor 0.5.9 übersetzt wurden, ist diese höchstwahrscheinlich nicht im Detail wiederherstellbar. Ab 0.5.9 wird die Version des verwendeten Solidity-Compilers allerdings im Metadata-Hash angegeben, wodurch diese verfügbar wäre und somit im dekompilierten Output angezeigt werden könnte. Die Version des verwendeten Solidity-Compilers ist prinzipiell auch für die Wiederherstellung des Schlüsselworts `payable` relevant. Dieses Keyword wurde ab der Version 4 eingeführt und könnte für neuere Contracts reproduziert werden, falls der Check `require(0 == msg.value);` nicht die erste Zeile einer Funktion darstellt. Nur dann darf die Funktion einen ETH-Wert entgegennehmen [27].

Bis jetzt wurden Probleme identifiziert, welche voraussichtlich mit weiterer Forschung behoben bzw. unter Anwendung einer bestimmten Heuristik zumindest angenähert werden können. Bei gewissen Spracheigenschaften ist die Möglichkeit zur Wiederherstellung allerdings ziemlich unwahrscheinlich. Diese werden im Folgenden unter Anführung von Gründen beschrieben:

- Contractnamen
- Variablennamen
- Unterscheidung von bestimmten Schlüsselwörtern
- Sicherheitscheck-Unterscheidung: Compiler oder Entwickler
- Modifier

Da die Namen von Contracts und Variablen anhand des EVM-Bytecodes nicht wiederhergestellt werden können, wurde die Definition dieser Eigenschaften gänzlich weggelassen, um den Fokus voll und ganz auf die Wiederherstellung der Funktionalität zu legen. Um nun einen kompilierbaren Contract zu reproduzieren, müssten diese Eigenschaften hinzugefügt werden. Über den Bytecode hat man jedoch keine Kenntnis über die Nomenklatur, weshalb man auch für Contracts fortlaufende Namen vergeben müsste, so wie dies bereits für Storage- und lokale Variablen umgesetzt wird.

Bei der Darstellung von Funktionen ist die Unterscheidung der einzelnen Schlüsselwörter nicht möglich. Aus dem EVM-Bytecode lässt sich keine Information ableiten, ob eine

öffentliche Funktion ursprünglich mit `public` oder `external` markiert war. Selbiges gilt für die Unterscheidung von `private` oder `internal` bei privaten Funktionen, welche ohnehin nur mit einer Heuristik angenähert werden können. Auch die Markierungen `view` und `pure` lassen sich auf Basis des Bytecodes nicht reproduzieren.

Wie in dieser Arbeit gezeigt wurde, kann zwar zwischen `assert` und `require` unterschieden werden, jedoch ist es vermutlich nicht möglich zu entscheiden, welche Checks bereits im ursprünglichen Solidity-Code enthalten waren und welche nachträglich vom Compiler eingefügt wurden. Nichtsdestotrotz ändern diese Überprüfungen nichts an der Semantik, da sie nur für höhere Stabilität im Contract sorgen. Da sie außerdem in der aktuellen Form kompiliert werden können, muss zwischen den beiden Arten nicht notwendigerweise unterschieden werden.

Ähnliches gilt für *Modifier*. Der Bytecode besitzt keine Information über diese Sprachelemente, da die `require`-Checks, welche mithilfe von `modifier` umgesetzt wurden, vom Compiler direkt in die jeweilige Funktion aufgenommen werden. Auch in diesem Fall ist aufgrund der äquivalenten Semantik jedoch keine separate Betrachtung notwendig.

Abschließend ist zu erwähnen, dass anhand des EVM-Bytecodes der Großteil des ursprünglichen Solidity-Codes wiederhergestellt werden kann. Aus den obigen Gründen ist es jedoch nicht möglich, bei der Dekompilierung von EVM-Bytecode syntaktische Gleichheit zu erzielen, also den ursprünglichen Code vollständig identisch wiederherzustellen. Es besteht allerdings höchstwahrscheinlich die Möglichkeit, einen semantisch äquivalenten Contract zu reproduzieren, welcher sich gleich verhält wie der ursprünglich vom Entwickler implementierte Contract und vom Solidity-Compiler übersetzt werden kann. Um dies zu erreichen, bedarf es allerdings weiterer Forschung in dieser Materie.

KAPITEL 6

Einschränkungen

Soldec stellt keinen vollständigen Decompiler dar und kann aus EVM-Bytecode folglich keinen kompilierbaren Solidity-Code reproduzieren. Abschnitt 5.4 befasst sich ausführlich mit der Evaluierung des Prototyps und zeigt das Forschungspotenzial auf, welches für die vollständige Dekompilierung notwendig wäre. Zusammengefasst ergeben sich aus den obigen Erkenntnissen daher einige offene Forschungsfragen. Weiterführende Themen wären beispielsweise die Verbesserung der Erkennung von Kontrollstrukturen, damit keine ungewollten Sprünge im rückübersetzten Output verbleiben. Ferner können hinsichtlich Variablen- und Typreproduktion weitere Untersuchungen durchgeführt werden. Außerdem müssen bei der Wiederherstellung von Konstruktoren, privaten und Fallback-Funktionen effektivere Heuristiken entwickelt werden.

Eine weitere wesentliche Einschränkung dieser Arbeit ist, dass im Hinblick auf die Dekompilierung nicht alle Solidity-Features betrachtet wurden. So wurde beispielsweise den Konzepten Vererbung, Interfaces, abstrakte Contracts und Bibliotheken im Rahmen dieser Arbeit keine Beachtung geschenkt. Darüber hinaus wurde die Rückübersetzung einiger EVM-Bytecode-Attribute, wie Opcodes, Meta- und Calldata nicht behandelt. Auch hier könnte weiterführende Forschung betrieben werden.

Additional sei noch erwähnt, dass in der Ausgabe von Soldec die Namen der dekompilierten Contracts, welche ohne Kenntnis darüber nicht reproduzierbar sind, sowie das Schlüsselwort `contract` und die nachfolgenden gruppierenden, geschwungenen Klammern genauso wie die Variablen- und Eventdeklarationen der Einfachheit halber gänzlich weggelassen werden.

KAPITEL 7

Fazit

Diese Arbeit hat untersucht, inwieweit die Dekompilierung von EVM-Bytecode mit traditionellen Methoden vorangetrieben werden kann. Zu diesem Zweck war es einerseits notwendig, die EVM-Bytecode-Struktur von aktuellen Smart Contracts zu analysieren. Eine wesentliche Erkenntnis dabei war, dass der Bytecode bei der Dekompilierung nicht im Gesamten betrachtet werden darf, sondern in separate Teile getrennt werden muss, welche entweder für die Dekompilierung nicht von Bedeutung sind oder einzeln rückübersetzt werden müssen. Andererseits wurden im Rahmen dieser Arbeit jene Forschungsarbeiten analysiert, welche sich bereits ansatzweise mit der Dekompilierung von EVM-Bytecode beschäftigt haben. Diese Analyse gab Aufschluss darüber, dass die Dekompilierung mit traditionellen Methoden grundsätzlich möglich ist, jedoch weiterer Forschung bedarf. Auf Basis eines fortgeschrittenen Decompilers wurde anschließend ein eigener Prototyp entwickelt, welcher diesen Forschungsbedarf zum Teil abdeckt.

Die wohl essenziellste Erkenntnis dieser Diplomarbeit ist, dass der ursprüngliche Solidity-Code auf Grundlage des EVM-Bytecodes nicht eins zu eins wiederherstellbar ist. Für einige Eigenschaften, wie zum Beispiel Namen von Contracts bzw. Variablen, Modifier oder auch Schlüsselwörter von Funktionen fehlen die benötigten Informationen im Bytecode. Mittels Dekompilierung von EVM-Bytecode ist es daher nicht möglich, unter Garantie ein syntaktisches Äquivalent zu reproduzieren, welches dem Solidity-Code des ursprünglichen Contracts gleicht. Was jedoch durch diese Diplomarbeit nicht auszuschließen ist, ist semantische Gleichheit. Mithilfe von weiterführender Forschung ist es also eventuell möglich, einen kompilierbaren Contract zu reproduzieren, welcher das gleiche Verhalten aufweist wie der ursprüngliche Contract. Nicht vorhandene Informationen wie zum Beispiel Namen oder auch Modifier würden durch Platzhalter oder semantisch äquivalente Konstrukte ersetzt werden.

Abschließend bleibt festzuhalten, dass diese Arbeit dazu beigetragen hat, den Stand der Dekompilierung von EVM-Bytecode zu erweitern, sodass es grundsätzlich möglich ist, Solidity-ähnlichen-Code zu reproduzieren. Dadurch können Smart Contracts der

Ethereum-Blockchain hinsichtlich Funktionalität und Sicherheitsproblemen analysiert werden, womit in weiterer Folge ein Wissenszuwachs möglich ist, welcher bei der Verwendung der Ethereum-Technologie für die Entwicklung von sicheren Applikationen eingesetzt werden kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

ABI Application Binary Interface.

API Application Programming Interface.

BTC Bitcoin.

CFG Control Flow Graph / Kontrollflussgraph.

DApps Decentralized Applications.

ETH Ether.

EVM Ethereum Virtual Machine.

HLL High-Level Programming Language / Höhere Programmiersprache.

HTML Hypertext Markup Language.

LIFO Last In – First Out.

LOC Lines of Code.

P2P Peer-to-Peer.

PDF Portable Document Format.

PoA Proof-of-Authority.

PoS Proof-of-Stake.

PoW Proof-of-Work.

TPS Transaktionen pro Sekunde.

UTXO Unspent Transaction Ouput.

Literaturverzeichnis

[1] Bitcoin Project Website. https://bitcoin.org. [Accessed: 14.05.2019].

[2] Decompiler Design - Loop Analysis. http://www.backerstreet.com/decompiler/loop_analysis.php. [Accessed: 10.01.2020].

[3] Decompiler Design - Type Analysis. https://www.backerstreet.com/decompiler/type_analysis.php. [Accessed: 10.01.2020].

[4] Ethereum Function Signature Database. https://www.4byte.directory. [Accessed: 13.11.2019].

[5] Ethereum Project Website. https://www.ethereum.org. [Accessed: 14.05.2019].

[6] GitHub. https://github.com. [Accessed: 21.05.2019].

[7] Go Ethereum. https://geth.ethereum.org. [Accessed: 27.11.2019].

[8] LLL Compiler Documentation. https://lll-docs.readthedocs.io/en/latest/index.html. [Accessed: 27.11.2019].

[9] Mutan. https://github.com/obscuren/mutan. [Accessed: 26.11.2019].

[10] Python. https://www.python.org. [Accessed: 14.11.2019].

[11] Remix Documentation. https://remix-ide.readthedocs.io/en/latest/. [Accessed: 27.11.2019].

[12] Remix, Ethereum IDE. https://remix.ethereum.org. [Accessed: 27.11.2019].

[13] Serpent. https://github.com/ethereum/wiki/wiki/Serpent-%5BDEPRECATED%5D. [Accessed: 27.11.2019].

[14] Swarm. https://swarm-guide.readthedocs.io/en/latest/. [Accessed: 23.01.2020].

[15] Vyper. https://vyper.readthedocs.io. [Accessed: 26.11.2019].

[16] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (Second Edition)*. Pearson Education, Boston, MA, USA, 2007.

[17] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

[18] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *arXiv:1809.03981*, 2018.

[19] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 353–368, 2013.

[20] V. Buterin. A Next-Generation Smart Contract and Decentralized Application Platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2014. [Accessed: 10.05.2019].

[21] C. Cifuentes. *Reverse Compilation Techniques*. Queensland University of Technology, Brisbane, 1994.

[22] C. Cifuentes. Structuring Decompiled Graphs. In *International Conference on Compiler Construction*, pages 91–105. Springer, 1996.

[23] CISPA, Saarland University. teEther - Analysis and automatic exploitation framework for Ethereum smart contracts. `https://github.com/nescio007/teether`. [Accessed: 14.05.2019].

[24] Comae Technologies. Porosity. `https://github.com/comaeio/porosity`. [Accessed: 10.05.2019].

[25] ConsenSys. Mythril. `https://github.com/ConsenSys/mythril-classic`. [Accessed: 10.05.2019].

[26] M. di Angelo and G. Salzer. A Survey of Tools for Analyzing Ethereum Smart Contracts. `https://pdfs.semanticscholar.org/5fcd/6089a4973d3ddd7ca831b7129046c87f33c6.pdf`, 2019. [Accessed: 10.05.2019].

[27] Ethereum. Solidity Documentation. `https://solidity.readthedocs.io`. [Accessed: 27.11.2019].

[28] etherscan.io. EVM bytecode decompiler. `https://etherscan.io/bytecode-decompiler`. [Accessed: 14.11.2019].

[29] etherscan.io. Verified Contracts. `https://etherscan.io/contractsVerified`. [Accessed: 14.11.2019].

[30] ethervm.io. Ethereum Virtual Machine Opcodes. https://ethervm.io. [Accessed: 13.11.2019].

[31] ethervm.io. Online Solidity Decompiler. https://ethervm.io/decompile. [Accessed: 13.11.2019].

[32] I. Guilfanov. Decompilers and beyond. *Black Hat USA*, 2008.

[33] T. E. Hybel. DSol. https://github.com/tehybel/DSol-decompiler. [Accessed: 10.05.2019].

[34] T. E. Hybel. Decompilation of Ethereum smart contracts. Master's thesis, Universitas Arhusiensis, Aarhus, Dänemark, 2018.

[35] J. Davis, The New Yorker. The Crypto-Currency. https://www.newyorker.com/magazine/2011/10/10/the-crypto-currency. [Accessed: 10.01.2020].

[36] S. King and S. Nadal. Ppcoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. *self-published paper, August*, 19, 2012.

[37] J. Krupp and C. Rossow. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333. USENIX Association, 2018.

[38] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[39] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[40] melonport. Oyente. https://github.com/melonproject/oyente. [Accessed: 10.05.2019].

[41] B. Mueller. Smashing Ethereum Smart Contracts for Fun and Real Profit. In *The 9th annual HITB Security Conference*, 2018.

[42] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008. [Accessed: 10.05.2019].

[43] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies*. Princeton University Press, Princeton, New Yersey, USA, 2016.

[44] K. Olmos and E. Visser. Strategies for Source-to-Source Constant Propagation. *Electronic Notes in Theoretical Computer Science*, 70(6):156–175, 2002.

[45] Online Tools. Keccak-256 online hash function. https://emn178.github.io/online-tools/keccak_256.html. [Accessed: 14.05.2019].

[46] A. Santander. Deconstructing a Solidity Contract. https://blog.zeppelin.solutions/deconstructing-a-solidity-contract-part-i-introduction-832efd2d7737. [Accessed: 14.05.2019].

[47] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[48] Smart Contract Research at USYD. Vandal. https://github.com/usyd-blockchain/vandal. [Accessed: 10.05.2019].

[49] M. Suiche. Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode. *DEF CON*, 25:11, 2017.

[50] N. Szabo. Smart Contracts. *Virtual School*, 1994.

[51] teamnsrg. Erays. https://github.com/teamnsrg/erays. [Accessed: 10.05.2019].

[52] M. Wohrer and U. Zdun. Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.

[53] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. https://github.com/ethereum/yellowpaper, 2014. [Accessed: 10.05.2019].

[54] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *NDSS*, 2015.

[55] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1371–1385. USENIX Association, 2018.